

April 27 - 30, 2015

*Salishan Lodge
Gleneden Beach, Oregon*

Fault Tolerance for Numerical Library Routines

Jack Dongarra

University of Tennessee
Oak Ridge National Laboratory
University of Manchester



Overview

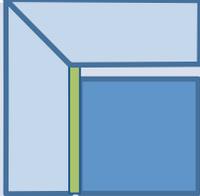
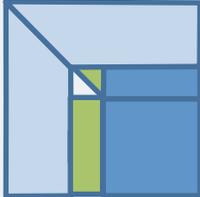
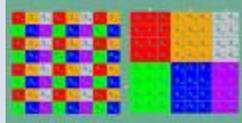
- **Can we easily generate LA library quality software that provides protection from faults?**
- **Can we protect against both hard and soft failures?**
- **Are the overheads reasonable?**



Problem Definition

- **Dense Linear Algebra**
 - **Direct methods: LU, Cholesky, QR, Singular values, and eigenvalue problems, ...**
- **Interested in understanding how we could enable this software to be fault tolerant without a total rewrite of the library.**
- **System Errors**
 - **Hard Error (single & multiple)**
 - Fail stop model, process completely and definitely stops
 - **Silent (soft) Error (single & multiple)**
 - Detect and correct say bit flips
- **Platforms/Applications**
 - **Distributed memory system**
 - Eg. ScaLAPACK and DPLASMA, with MPI

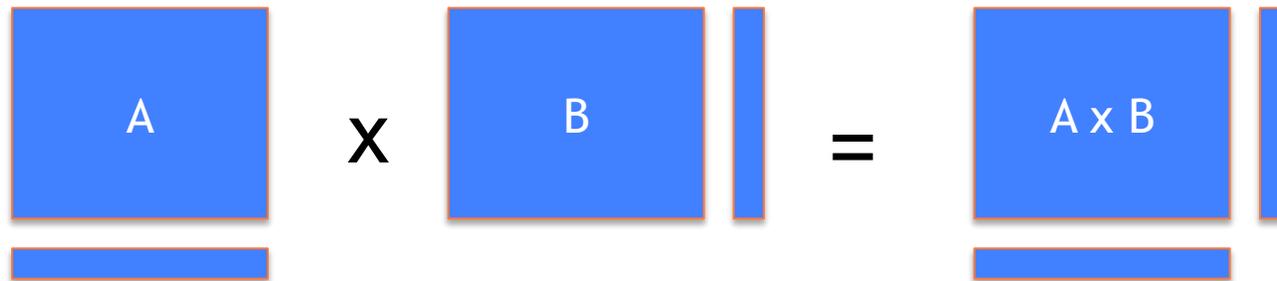
Last Generations of DLA Software

Software/Algorithms follow hardware evolution in time		
LINPACK (70's) (Vector operations)		Rely on - Level-1 BLAS operations
LAPACK (80's) (Blocking, cache friendly)		Rely on - Level-3 BLAS operations
ScaLAPACK (90's) (Distributed Memory)		Rely on - PBLAS Mess Passing

Algorithm Based Fault Tolerance (ABFT)

KH Huang & Jacob Abraham, ABFT for Matrix Operations, IEEE Trans. Computers. 1984;

- *Matrix extended to contain additional information.*
 - *Extra column or row contains checksum.*
- *Algorithm designed to operate on the data and the encoded checksum.*
- *Checksum invariant during the course of the algorithm.*
- *No checkpoint needed.*



$$\begin{bmatrix} A \\ G^T A \end{bmatrix} \times \begin{bmatrix} B & BG \end{bmatrix} = \begin{bmatrix} AB & ABG \\ G^T AB & G^T ABG \end{bmatrix}$$

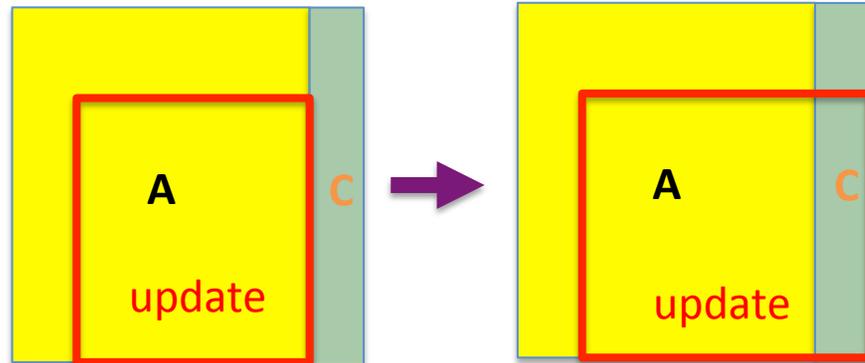
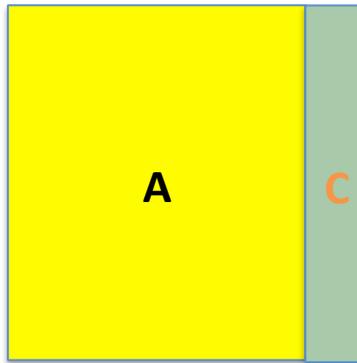
G^T  $G^T A$ and BG are the check sums.

ABFT Idea

- C matrix contains a checksum (e.g. row summations) of A
- Checksums remain mathematically invariant!

$$C_i = \sum_j A_{ij}$$

The Same algorithm updates both the trailing matrix AND the checksums



Here the generate, G, is a column of all one's.

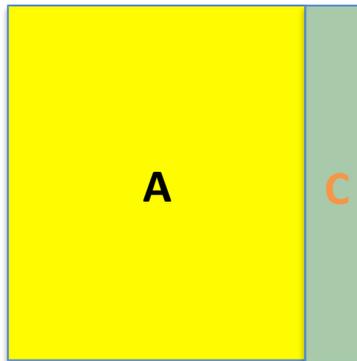
$$G^T = (1, 1, \dots, 1)$$

$$Update\left(\sum_j A_{ij}\right) = \sum_j Update(A_{ij})$$

ABFT Idea

- C matrix contains a checksum (row summations) of A
- Checksums introduce redundancy (resilience) to the algorithm

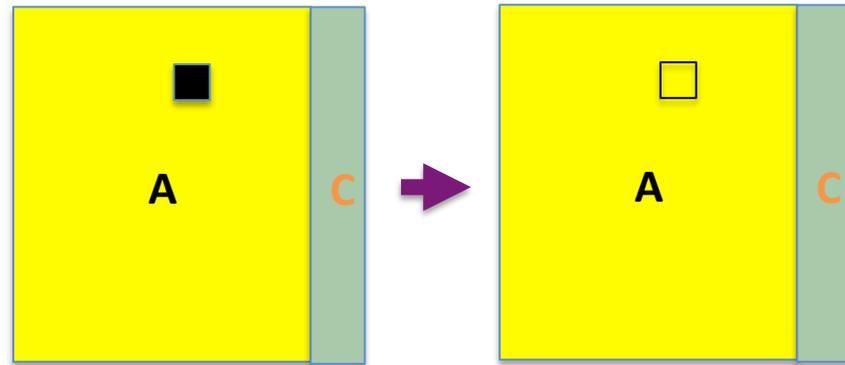
$$C_i = \sum_j A_{ij}$$



Here the generate, G, is a column of all one's.

$$G^T = (1, 1, \dots, 1)$$

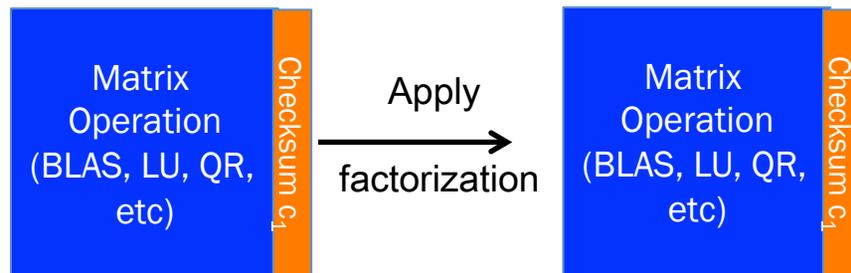
$$A_{ij} = C_i - \sum_{k \neq j} A_{ik}$$



In case of failure, checksum inversion allows to restore the missing value

Detection & Local Correction through ABFT Strategy

- Before the factorization starts, a checksum is taken
- Column(s) appended to the matrix
- Factorization proceeds with the augmented matrix
- Check the results after completion or use checksum to recover missing information.
- Checksum invariance



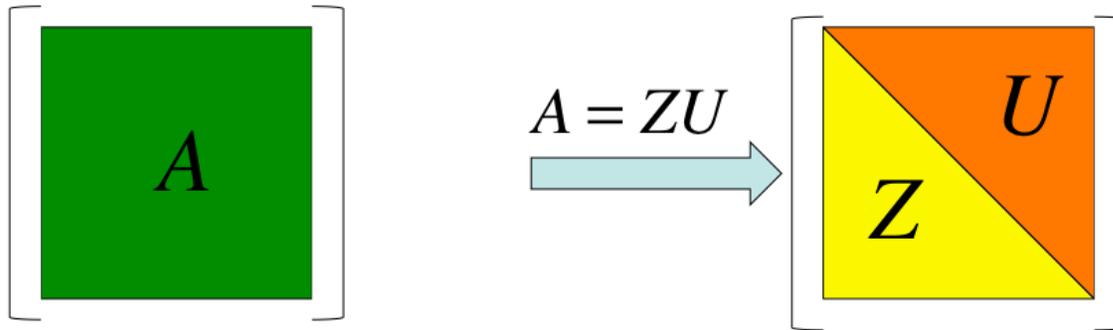
$$c_1 = Ae_1, \text{ where } e_1 = (1, 1, \dots, 1)^T$$

(A, c_1) Undergoes factorization, QR in this case

$$(\tilde{A}, \tilde{c}_1) \text{ Producing } (QR, \tilde{c}_1)$$

$$QR e_1 \stackrel{?}{=} \tilde{c}_1 \text{ Check Results (up to round off error)}$$

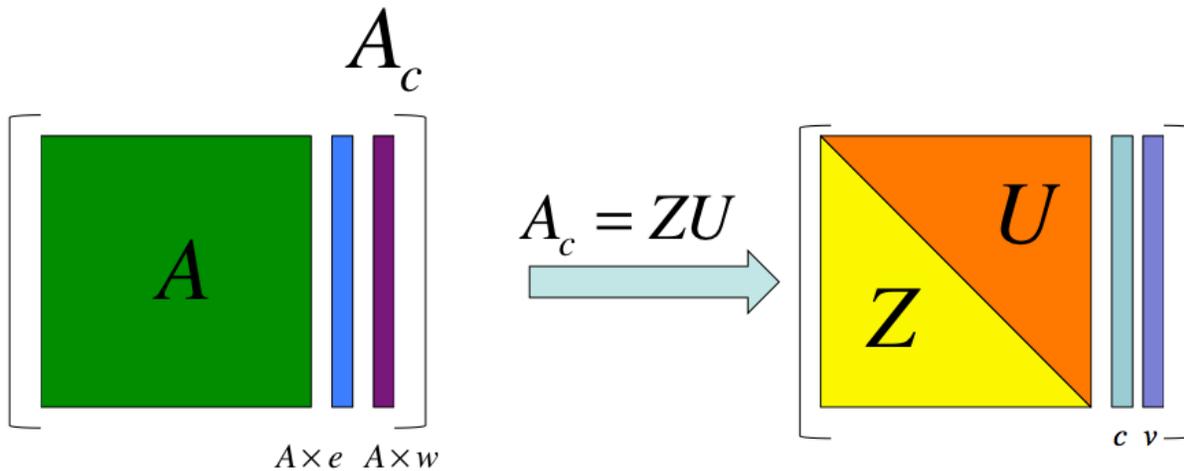
Checksum in ABFT



Generator, $G = (e, w)$

$$e = \begin{pmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{pmatrix} \quad w = \begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{pmatrix}$$

w_j : random number



$$A_c = (A, Ae, Aw)$$

$$A_c = Z(U, c, v)$$

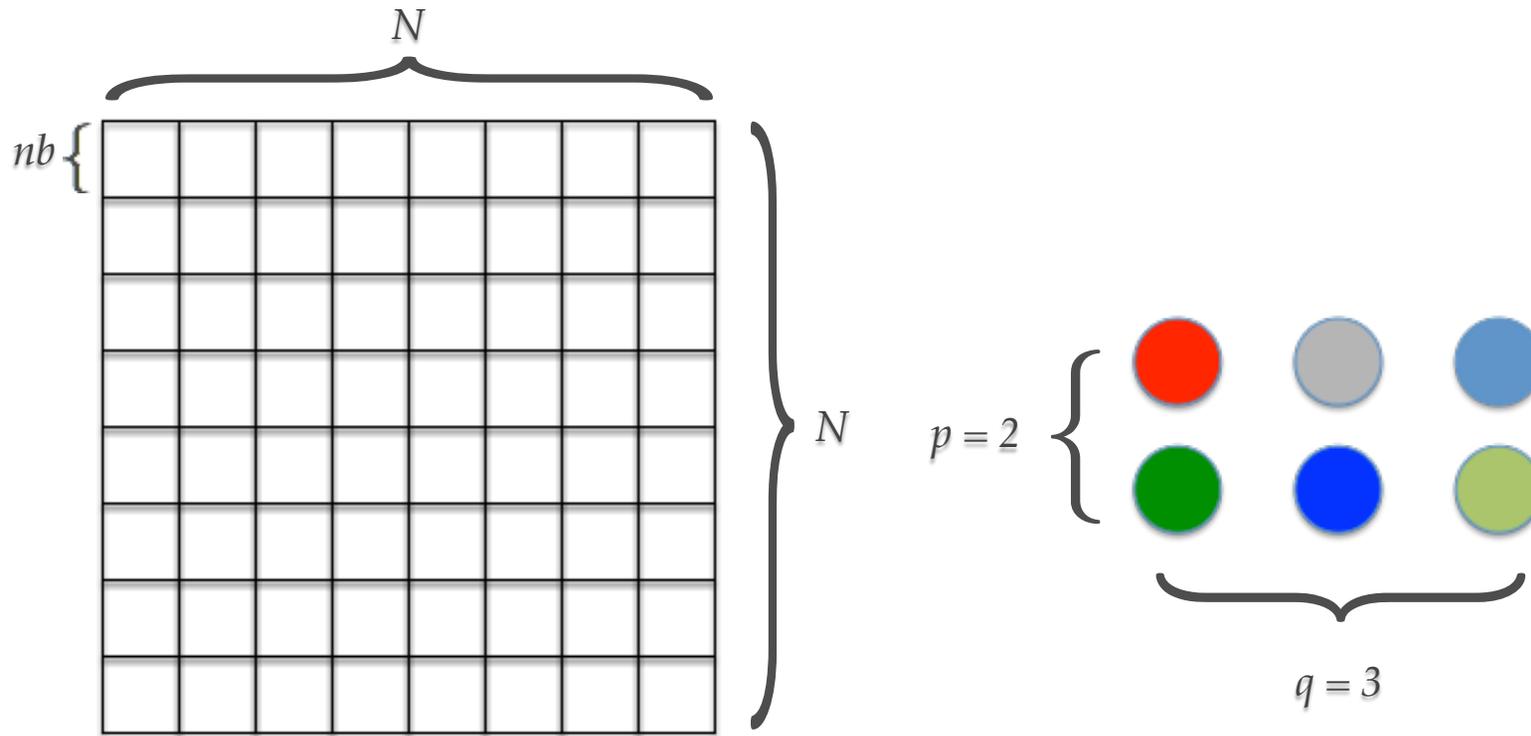
$$c = Ue$$

$$v = Uw$$

We are using random numbers for the Generators.

Alan Edelman, 89: Condition number of random matrix $\log(n) + 1.5367$

With ScaLAPACK Data has a 2-D Block Cyclic Distribution

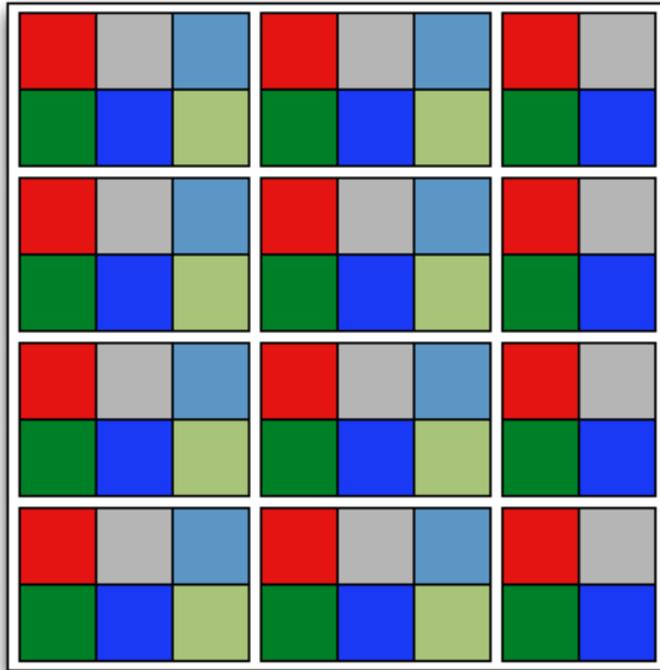


An $N \times N$ matrix partitioned into $nb \times nb$ blocks

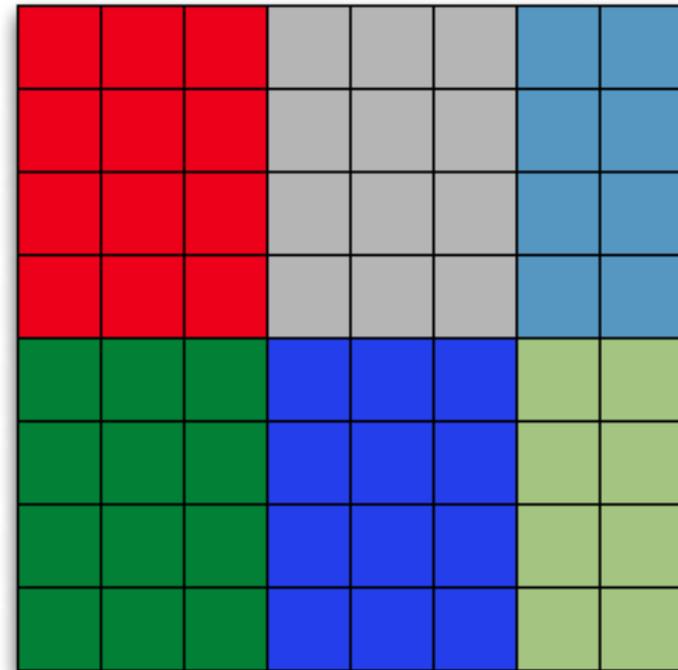
A $p \times q$ process grid ($p = 2, q = 3$)

Logical grid of processes

For Scalability Reasons 2-D Block Cyclic Data Distribution



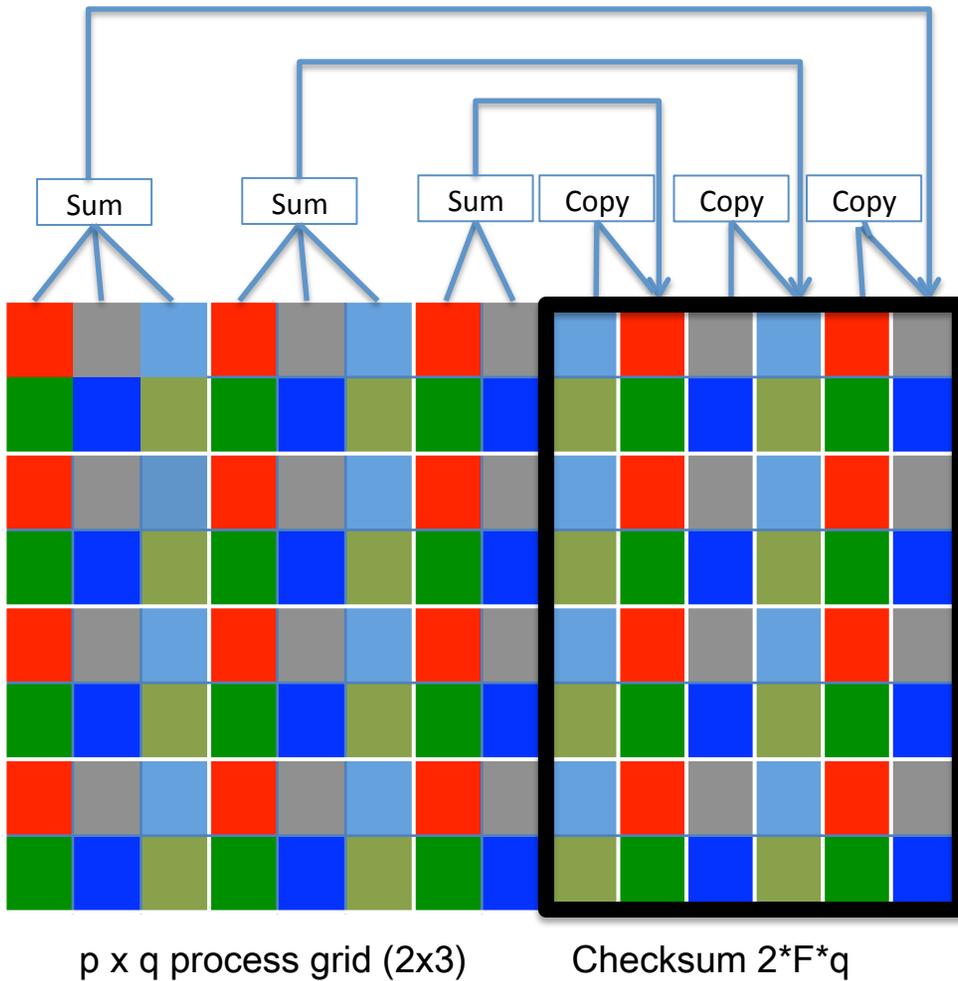
Matrix view



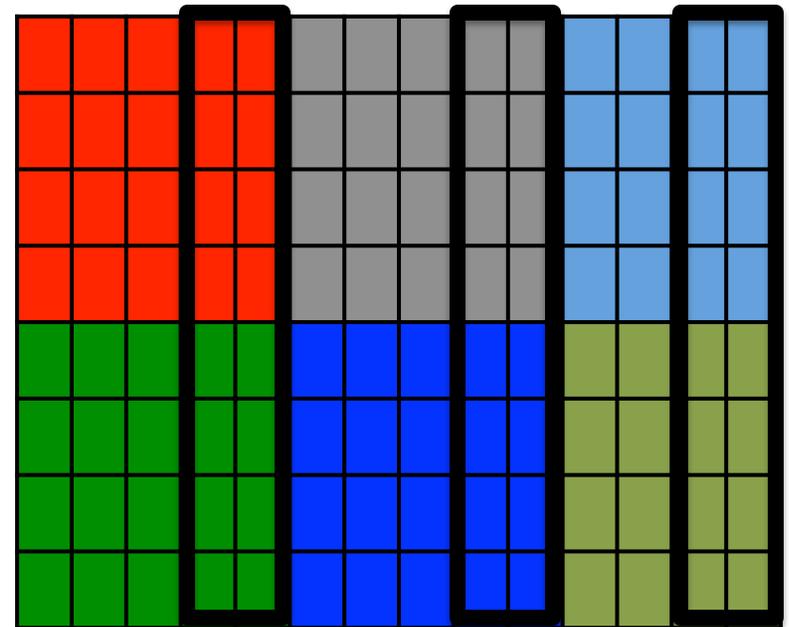
Process view

$p \times q$ process grid (2x3)

2-D Block Cyclic Data Distribution with Checksums

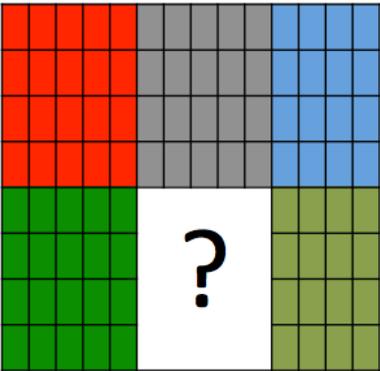
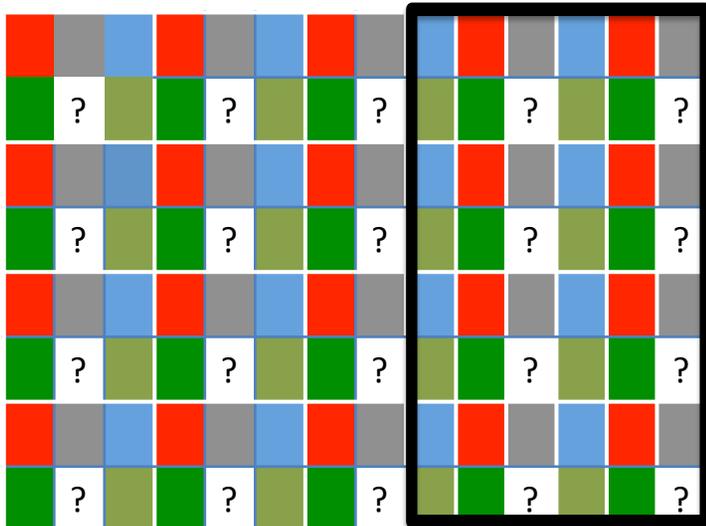
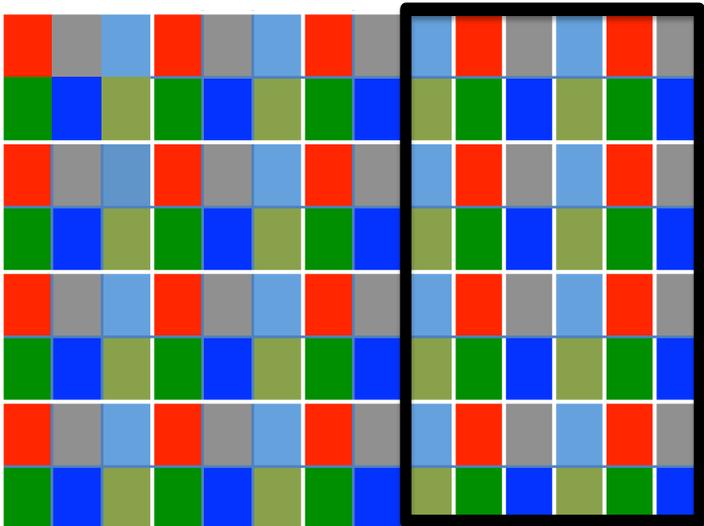


Checksums contained on active processes.

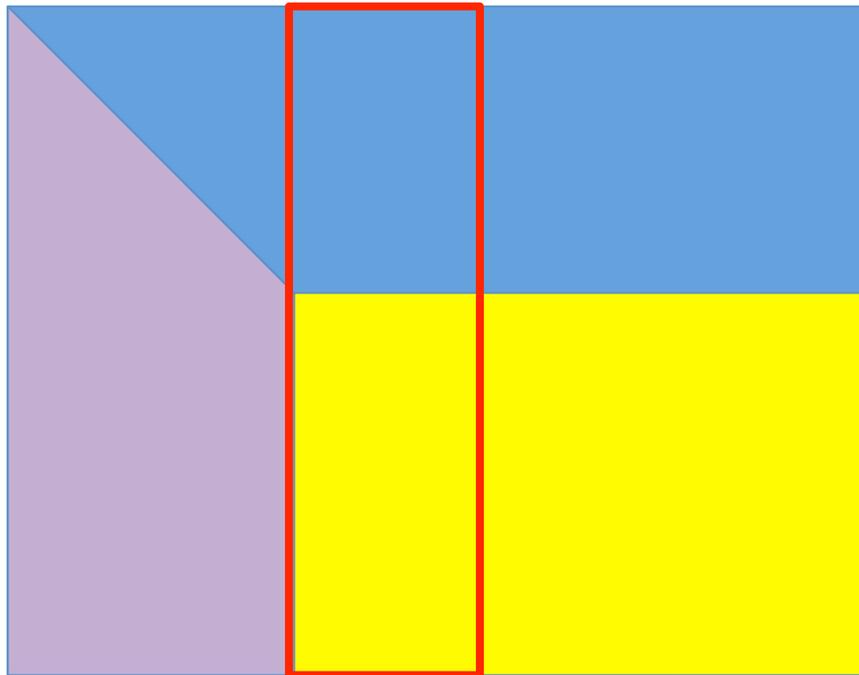


Matrix is extended with $2F$ columns for every q columns, here $F = 1$ and $q = 3$.
Checksum blocks are doubled to allow recovery when data and checksum are lost together.

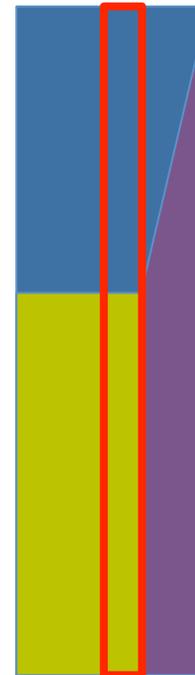
Failure Model



Group of Q panels

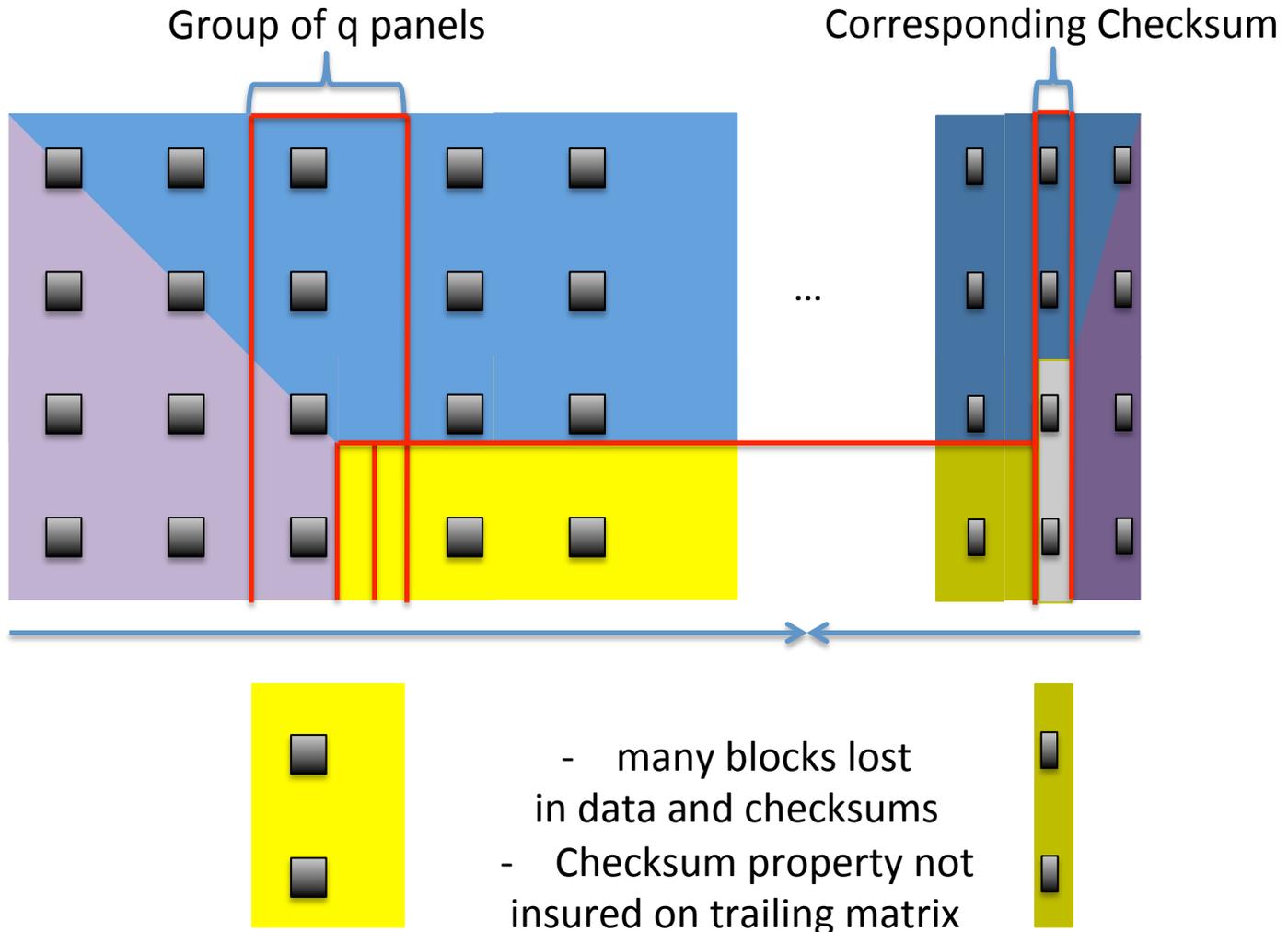


Corresponding Checksum



Make local copies of the group of q panels and the corresponding checksum

In case of Failure



ABFT Overheads on ScaLAPACK Like Implementation

Matrix $M \times N$, Blocks $m_b \times n_b$,
Process grid $p \times q$

F : maximum number of
simultaneous failures tolerated

Memory Overhead

$$O\left(\frac{F}{q} \times M \times N\right)$$

Matrix is extended with $2F$
columns every q columns

N.B. Usually $F \ll q$

Relative overheads in F/q

e.g. **2** simultaneous faults on **192x192**
process grid \Rightarrow 1% memory overhead

Computation Overhead

$$O\left(\frac{F}{q} \times M^3\right)$$

flops for the checksum update,
and

$$O(MN)$$

flops for the checksum creation.

**Less than 5% computational
overhead**

Kraken @ UTK



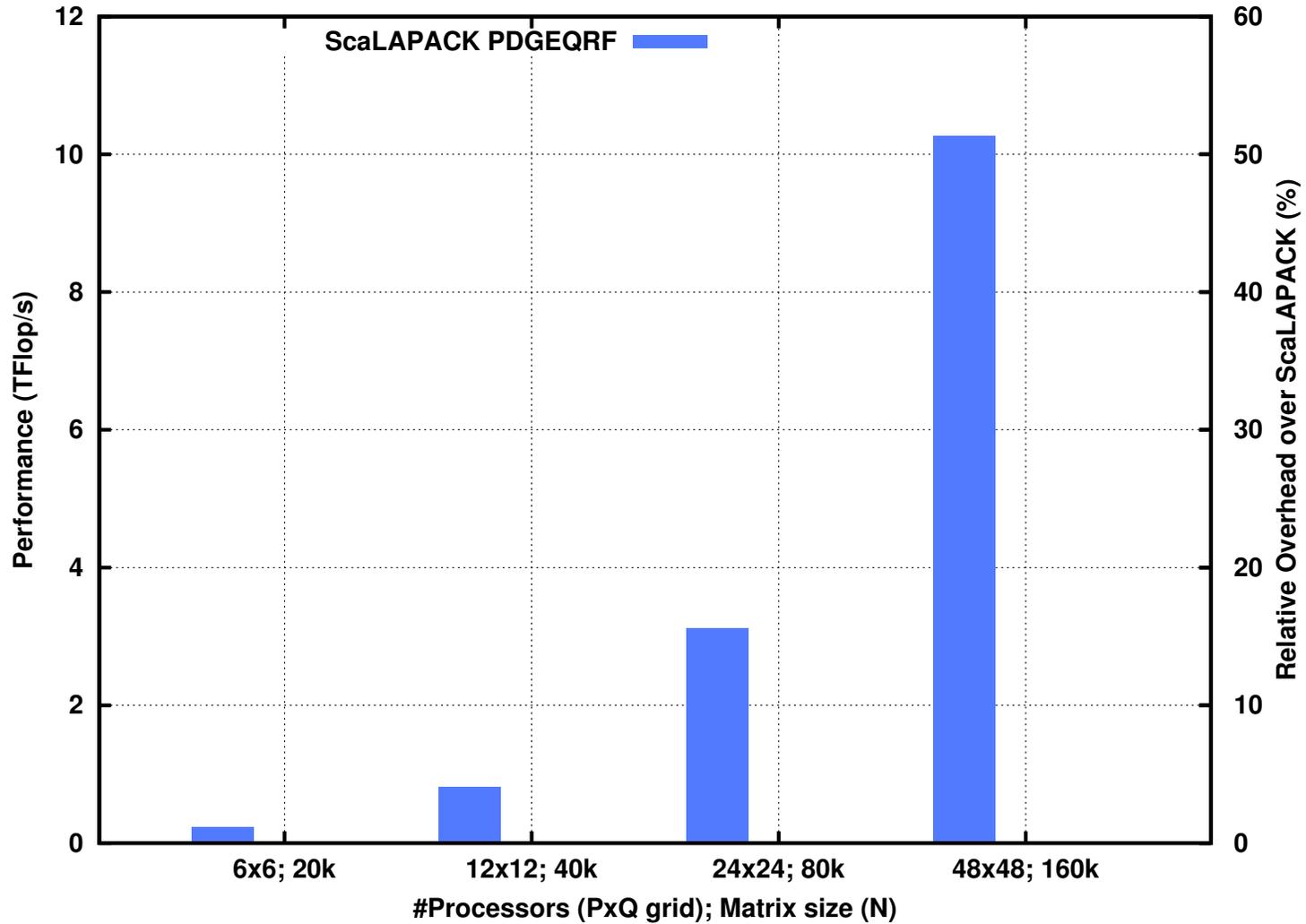
Kraken Cray XT5 system specifications:

- Cray Linux Environment (CLE) 3.1
- A peak performance of 1.17 PetaFLOP
- 112,896 compute cores
- 147 TB of compute memory
- A 3.3 PB raw parallel file system of disk storage for scratch space (2.4 PB available)
- 9,408 compute nodes

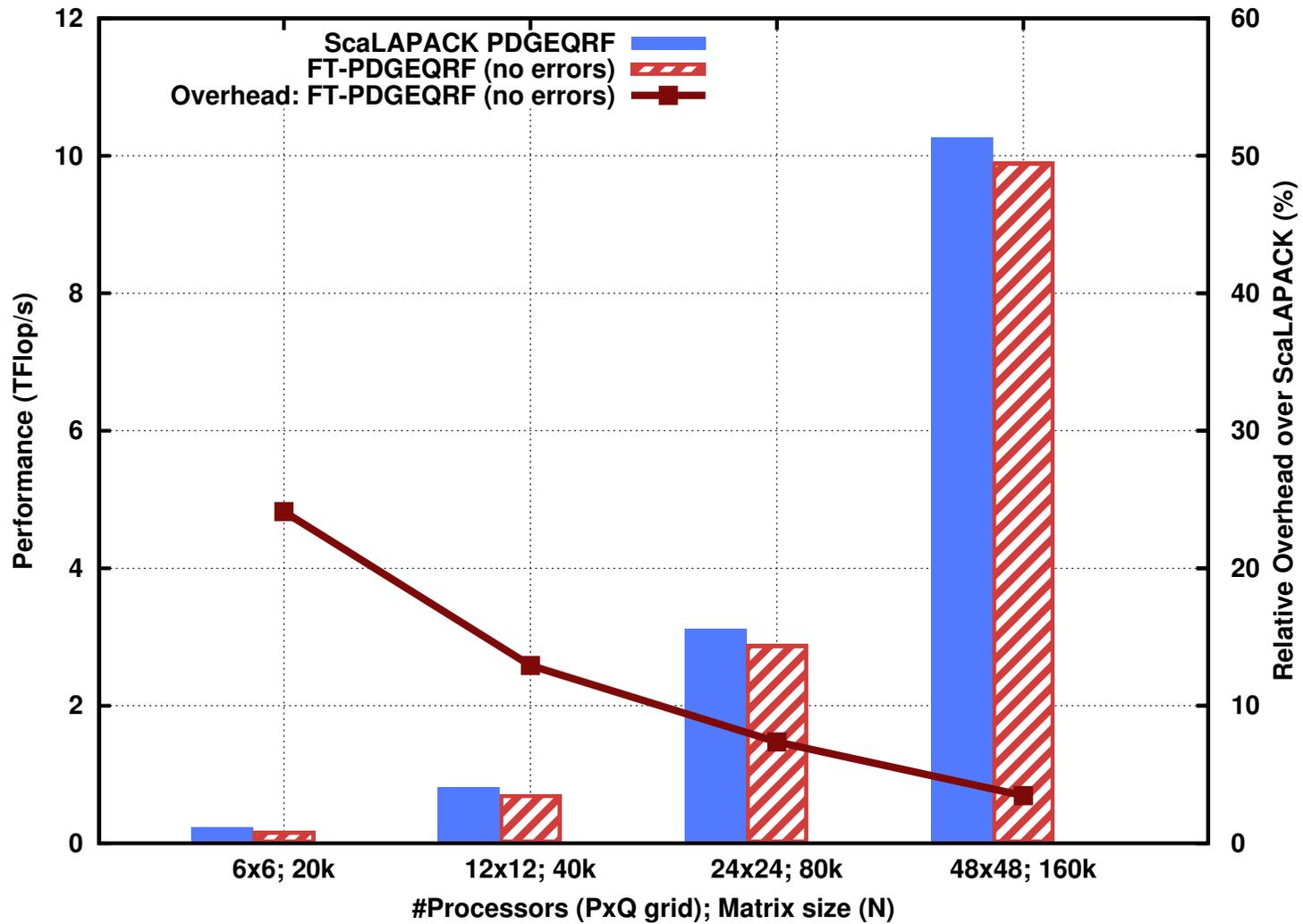
Each node has:

- Two 2.6 GHz six-core AMD Opteron processors (Istanbul)
- 12 cores
- 16 GB of memory
- Connection via Cray SeaStar2+ router

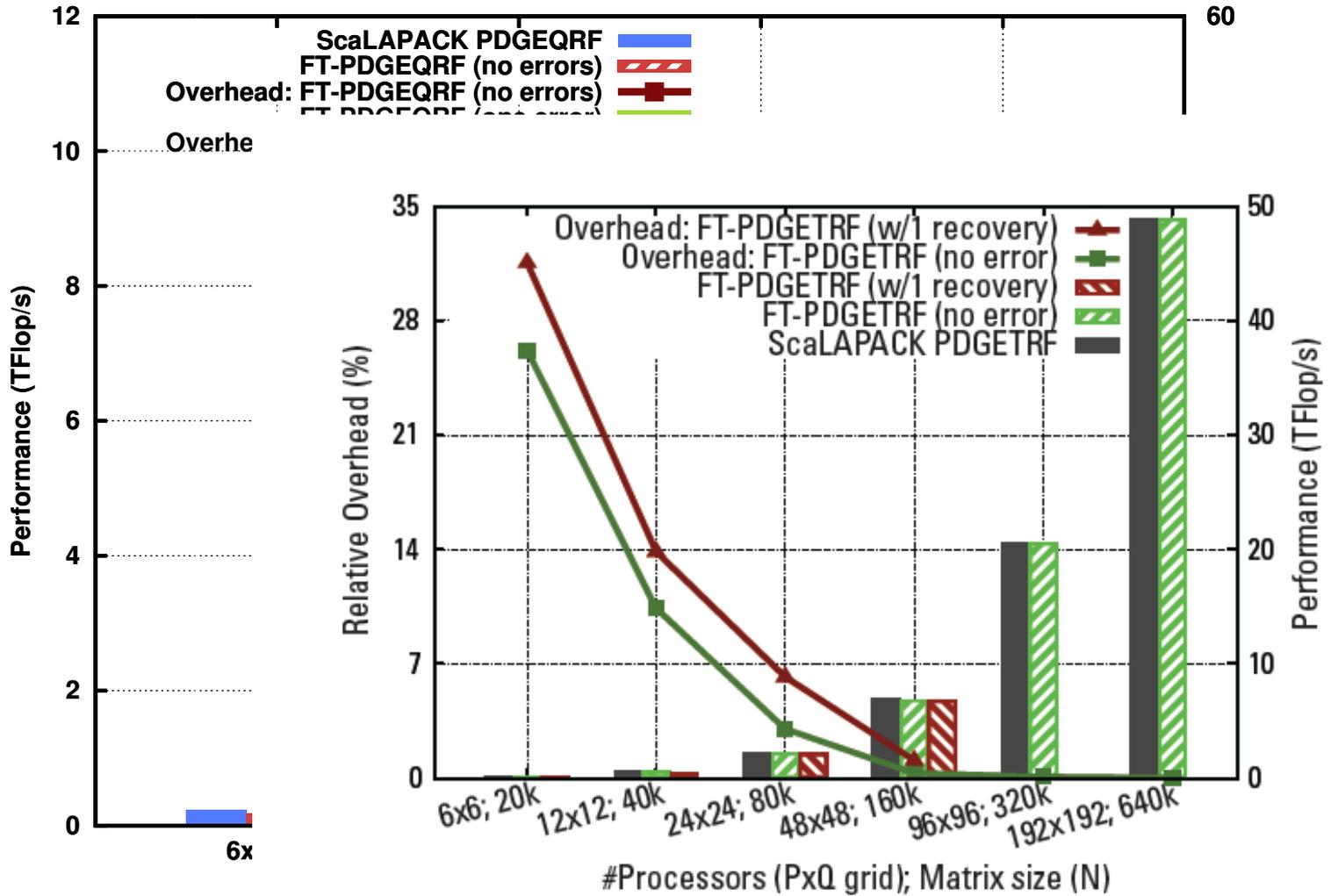
Performance for QR



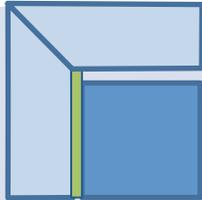
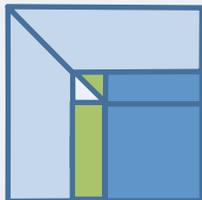
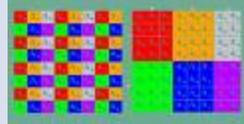
Performance for QR



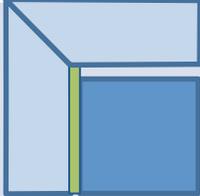
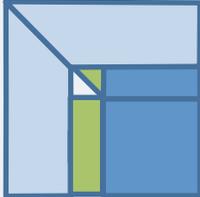
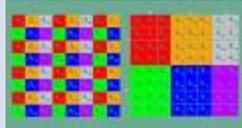
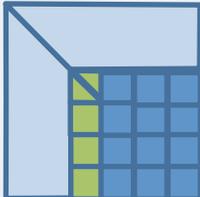
Performance for QR



Last Generations of DLA Software

Software/Algorithms follow hardware evolution in time		
LINPACK (70's) (Vector operations)		Rely on - Level-1 BLAS operations
LAPACK (80's) (Blocking, cache friendly)		Rely on - Level-3 BLAS operations
ScaLAPACK (90's) (Distributed Memory)		Rely on - PBLAS Mess Passing

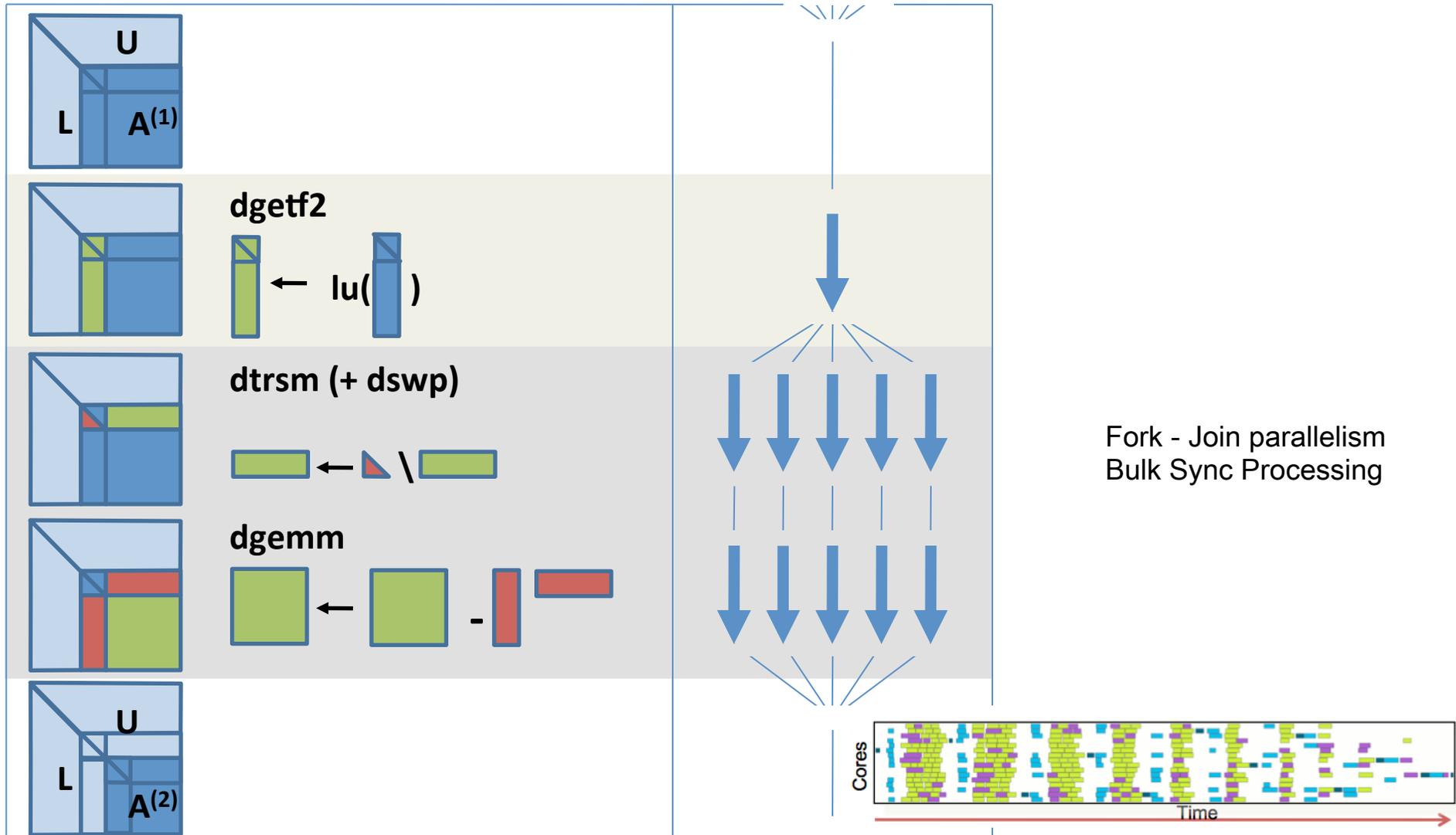
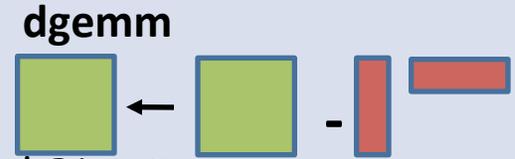
Current Generation of DLA Software

Software/Algorithms follow hardware evolution in time		
LINPACK (70's) (Vector operations)		Rely on - Level-1 BLAS operations
LAPACK (80's) (Blocking, cache friendly)		Rely on - Level-3 BLAS operations
ScaLAPACK (90's) (Distributed Memory)		Rely on - PBLAS Mess Passing
PLASMA New Algorithms (many-core friendly)		Rely on - a DAG/scheduler - block data layout - some extra kernels

Parallelization of LU and QR.

Parallelize the update:

- Easy and done in any reasonable software.
- This is the $2/3n^3$ term in the FLOPs count.
- Can be done efficiently with LAPACK+multithreaded BLAS

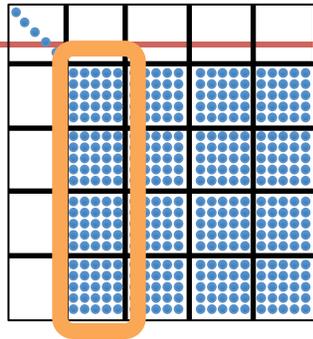




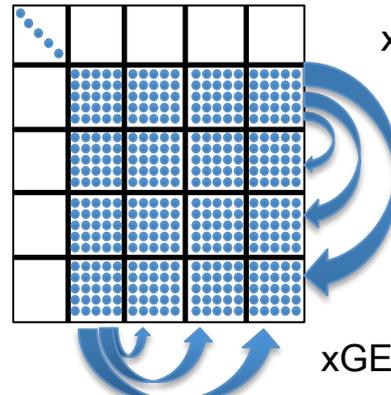
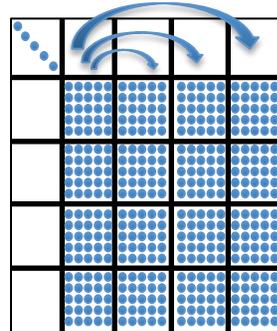
PLASMA LU Factorization

Dataflow Driven

Numerical program generates tasks and run time system executes tasks respecting data dependences.



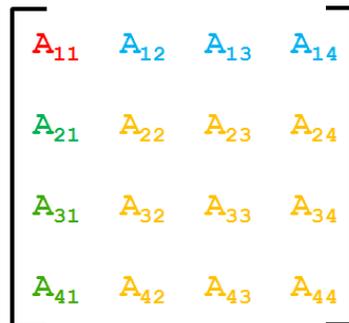
xTRSM



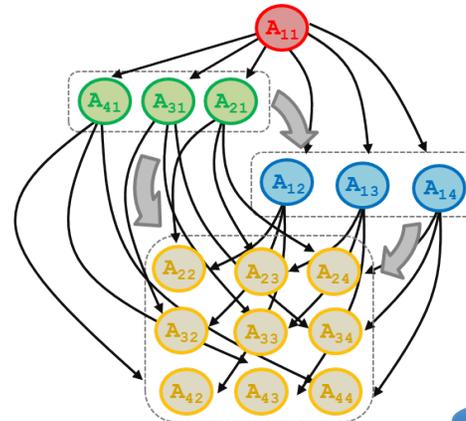
xGEMM

xGEMM

Sparse / Dense Matrix System



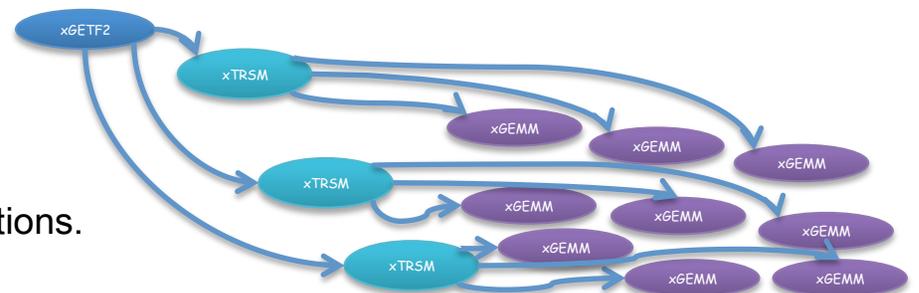
DAG-based factorization



- LU, QR, or Cholesky on small diagonal matrices
- TRSMs, QRs, or LUs
- TRSMs, TRMMs
- Updates (Schur complement) GEMMs, SYRKs, TRMMs

Batched LA

Each task, node in graph, is a matrix-matrix operations. Level 3 BLAS operation on tiles.



PLASMA - DAG Based Version

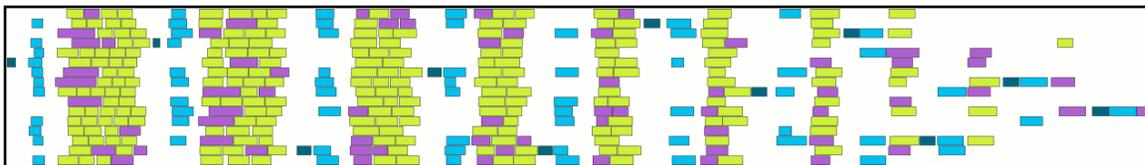
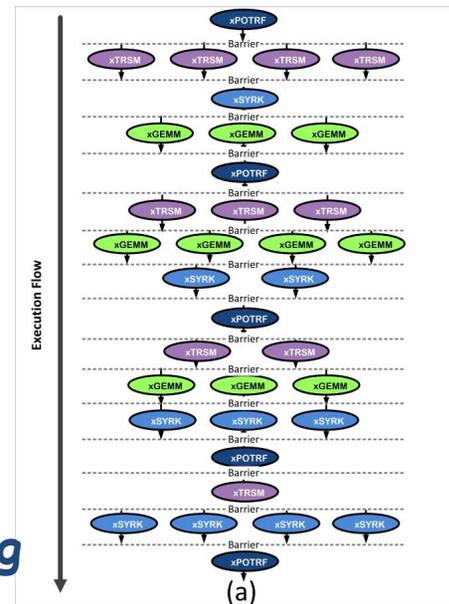
Objectives

- High utilization of each core
- Scaling to large number of cores
- Synchronization reducing algorithms

Methodology

- Dynamic DAG scheduling
- Explicit parallelism
- Implicit communication
- Fine granularity / block data layout

Arbitrary DAG with dynamic scheduling



Fork-join parallelism
Notice the synchronization penalty in the presence of heterogeneity.

PLASMA - DAG Based Version

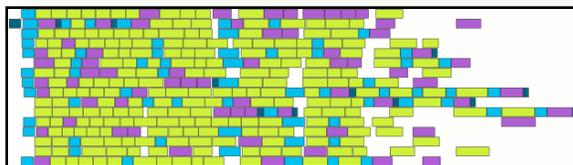
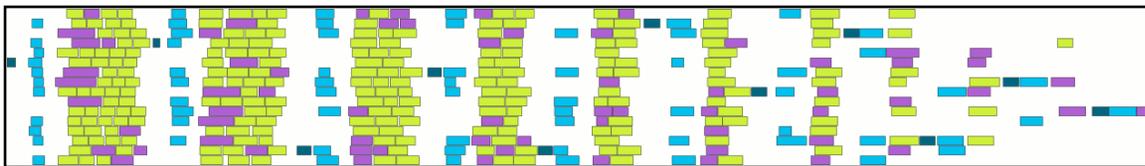
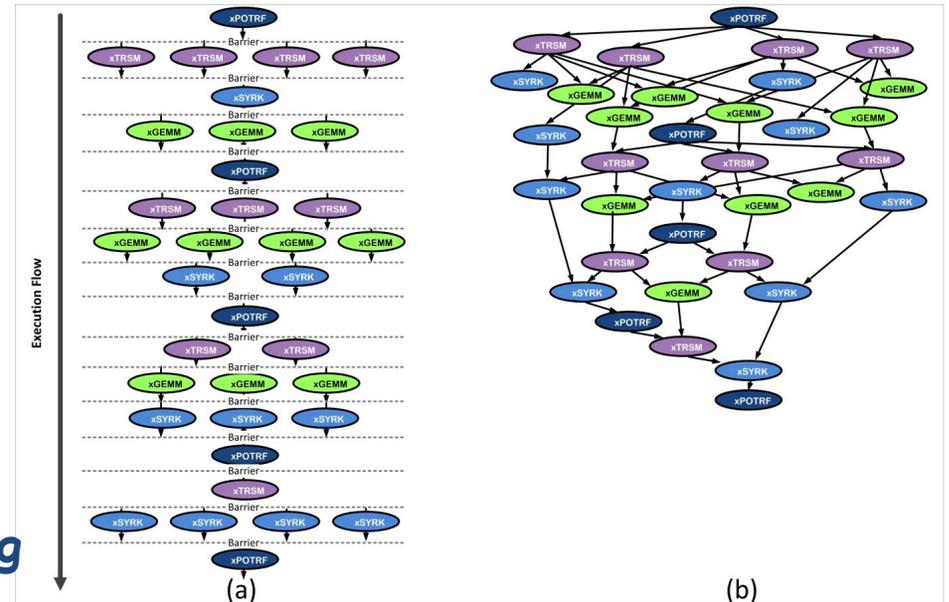
Objectives

- High utilization of each core
- Scaling to large number of cores
- Synchronization reducing algorithms

Methodology

- Dynamic DAG scheduling
- Explicit parallelism
- Implicit communication
- Fine granularity / block data layout

Arbitrary DAG with dynamic scheduling



DAG scheduled parallelism

Fork-join parallelism
Notice the synchronization penalty in the presence of heterogeneity.

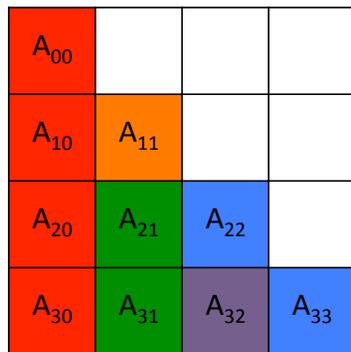
Motivation: Goal

- Failure model
 - **Soft error** (Silent Data Corruption): bit-flips, memory or processor registers
 - Here we focus on **soft errors** happening during computation
- Resilience / Fault Tolerance in dynamic task-based runtime
 - Implemented in PaRSEC, the runtime system for DPLASMA
 - Two levels of granularities and three mechanisms: Looking into DAG and task.
 - Case study on the Cholesky factorization

Introduction to PaRSEC

- Application representation

User's view



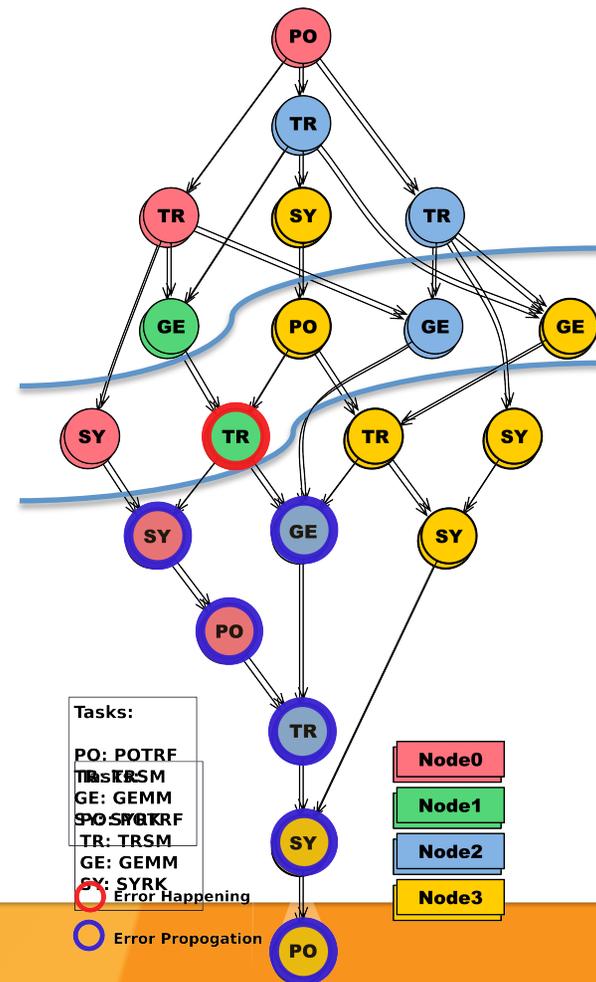
- Final result
- POTRF
- TRSM
- SYRK
- GEMM

```

FOR k = 0 .. SIZE - 1
  A[k][k], T[k][k] <- GEQRT( A[k][k] )
  FOR m = k+1 .. SIZE - 1
    A[k][k|Up, A[m][k], T[m][k] <-
      TSQRT( A[k][k|Up, A[m][k], T[m][k] )
  FOR n = k+1 .. SIZE - 1
    A[k][n] <- UNMQR( A[k][k|Low, T[k][k], A[k][n] )
    FOR m = k+1 .. SIZE - 1
      A[k][n], A[m][n] <-
        TSMQR( A[m][k], T[m][k], A[k][n], A[m][n] )
    
```

Silent error

Runtime's view



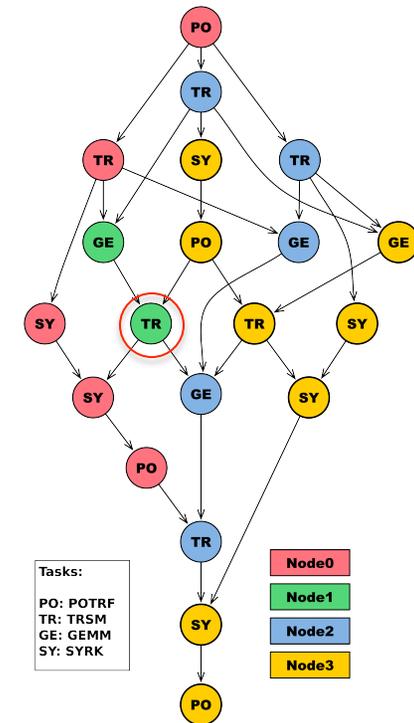
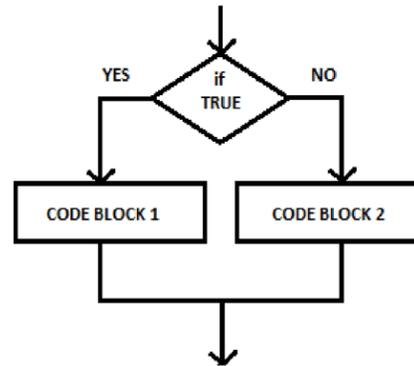
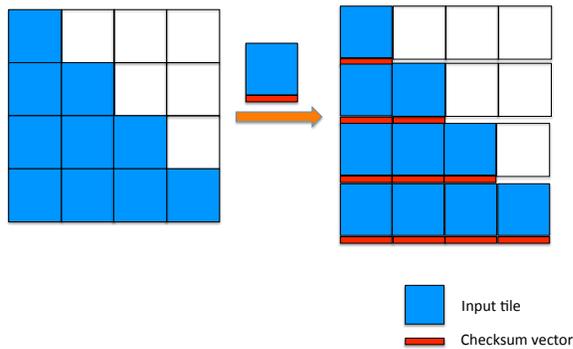
Detection & Local Correction through ABFT Strategy

- Implementation

(1) Attaching 2 checksum vectors to original data

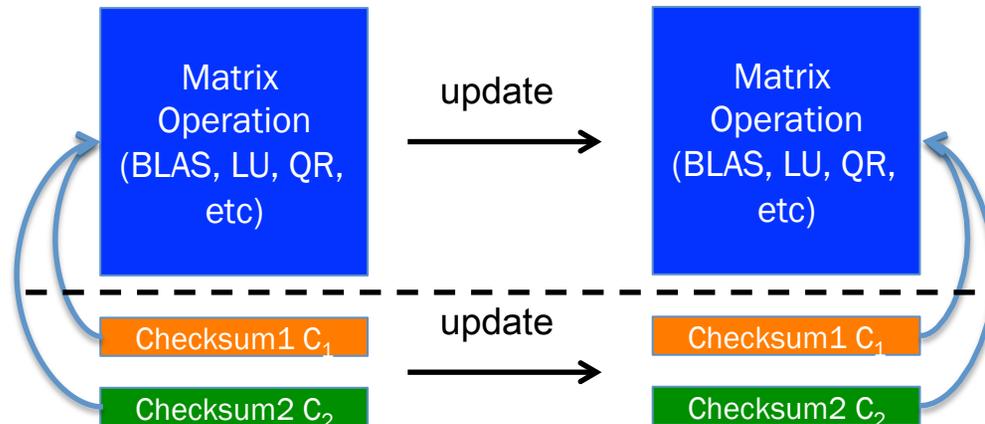
(2) Provide recovery scheme inside task

(3) Continue with the DAG execution



Detection & Local Correction through ABFT Strategy

- Checksum invariance



$$C_1 = Ag_1$$

$$C_2 = Ag_2$$

$$g_1 = (1, 1, \dots, 1)^T$$

$$g_2 = (1, 2, \dots, n)^T$$

(*because of round-off errors, a small tolerance is allowed)

- Single bit Detection & correction

- After update, single bit error happens at $A(i, j)$

$$\sum_{k=1}^n A(k, j) - C_1(j) = \gamma \Rightarrow \text{error is in column } j$$

$$\sum_{k=1}^n kA(k, j) - C_2(j) = i\gamma \Rightarrow \text{error is in row } i$$

$$A'(i, j) = A(i, j) - \gamma \Rightarrow \text{adding the difference to recover}$$

Analysis of Overhead

2. A single bit flip in task

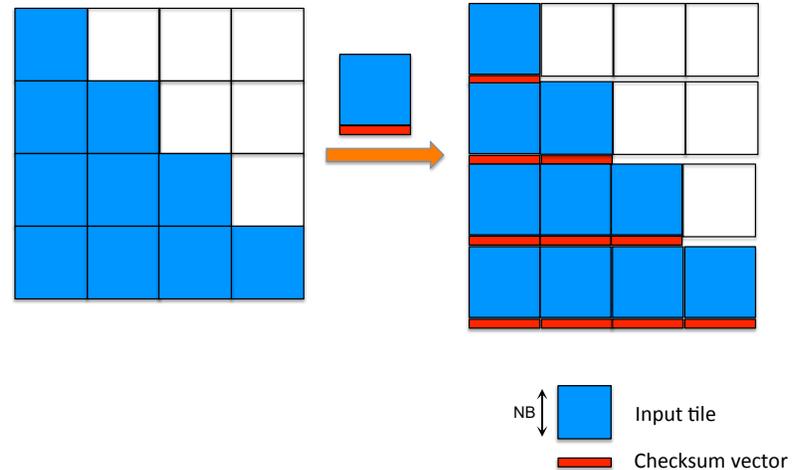
- 1) Applying ABFT method (avoid re-execution)

Attach 2 checksum vectors to every tile. Tile size is $NB \times NB$.

① Overhead (time)

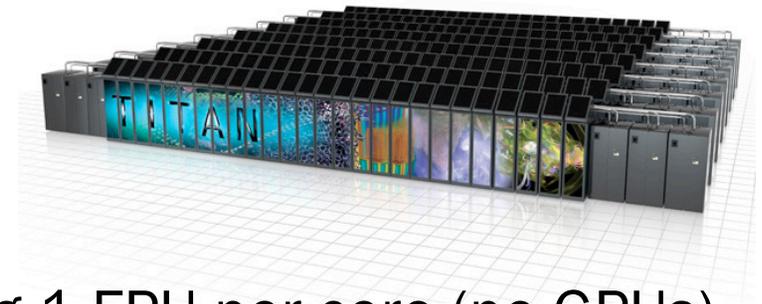
- Maintaining checksums: $(1 + \frac{2}{NB})^3 - 1$
- Detecting & correcting error: $\frac{1}{NB}$
- Total: $(1 + \frac{2}{NB})^3 - 1 + \frac{1}{NB}$

② Overhead (Storage) $\Rightarrow \frac{2}{NB}$

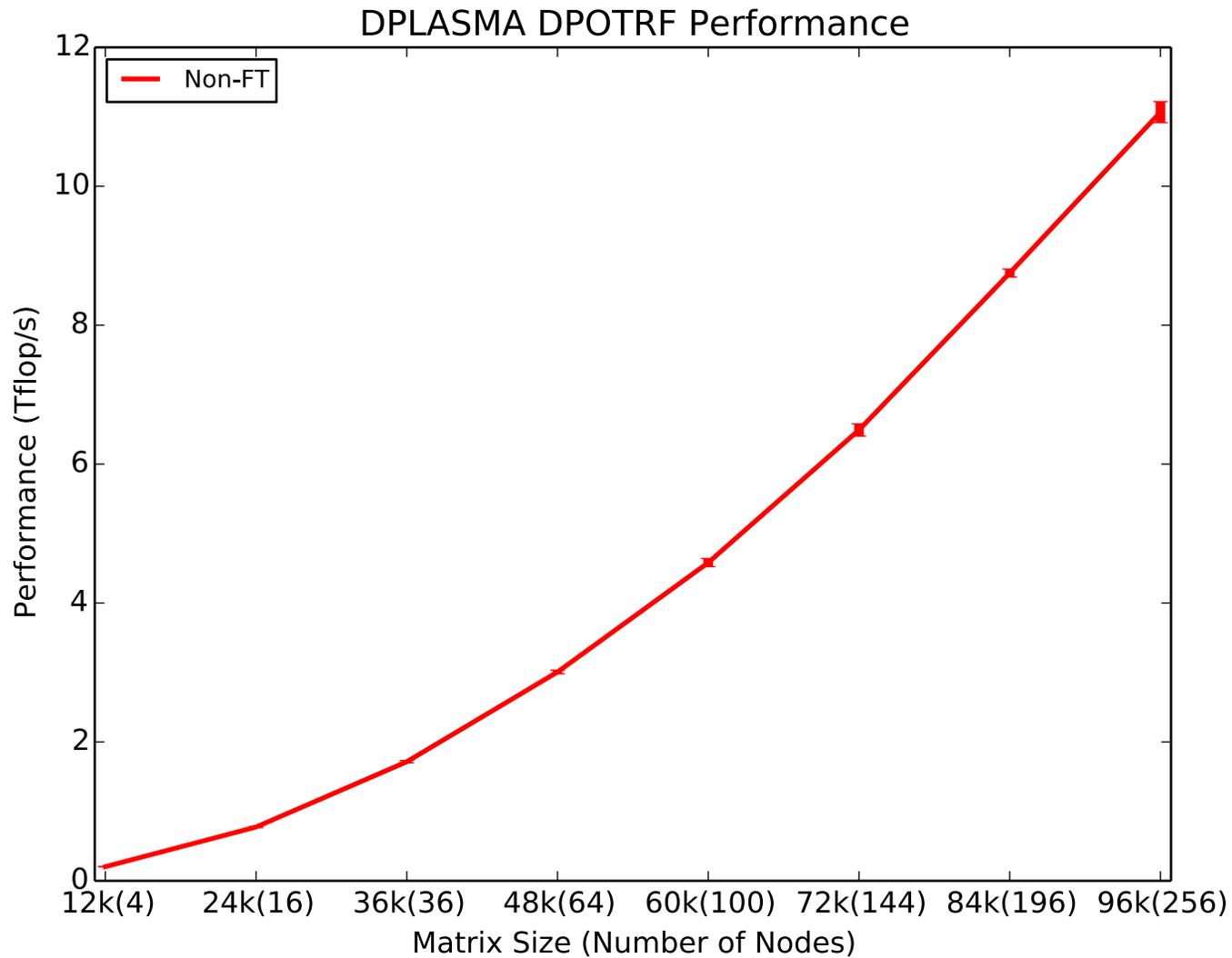


Experiment Platform

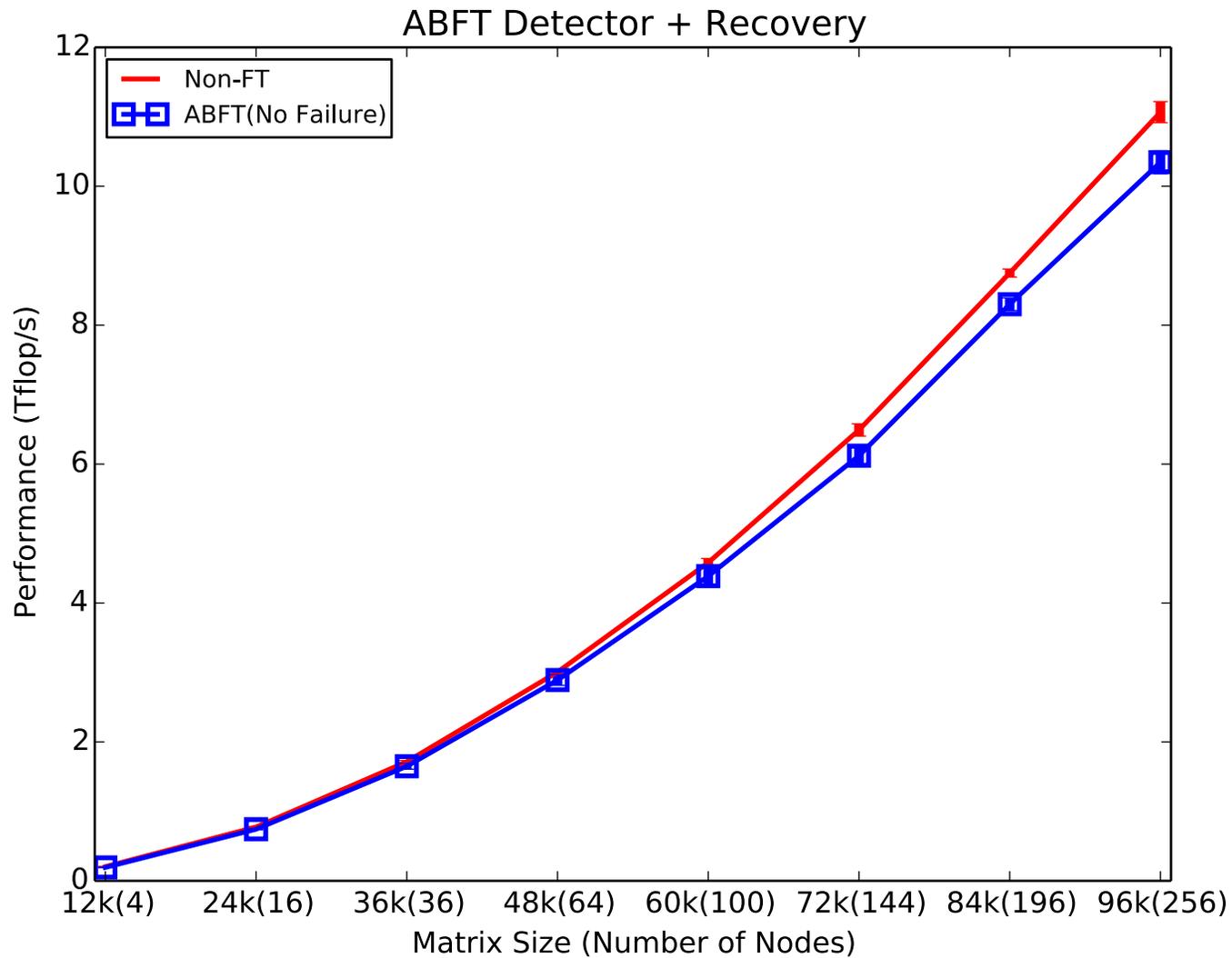
- Machine: Titan in ORNL
- CPU: AMD Opteron™ 6274 (Interlagos)
 - 16 Cores, 8 FPUs
 - We use 8 cores per CPU, ensuring 1 FPU per core (no GPUs)
 - Weak scaling experiments:
 - 6k x 6k matrix distributed on 1 node
 - Run up to 256 nodes



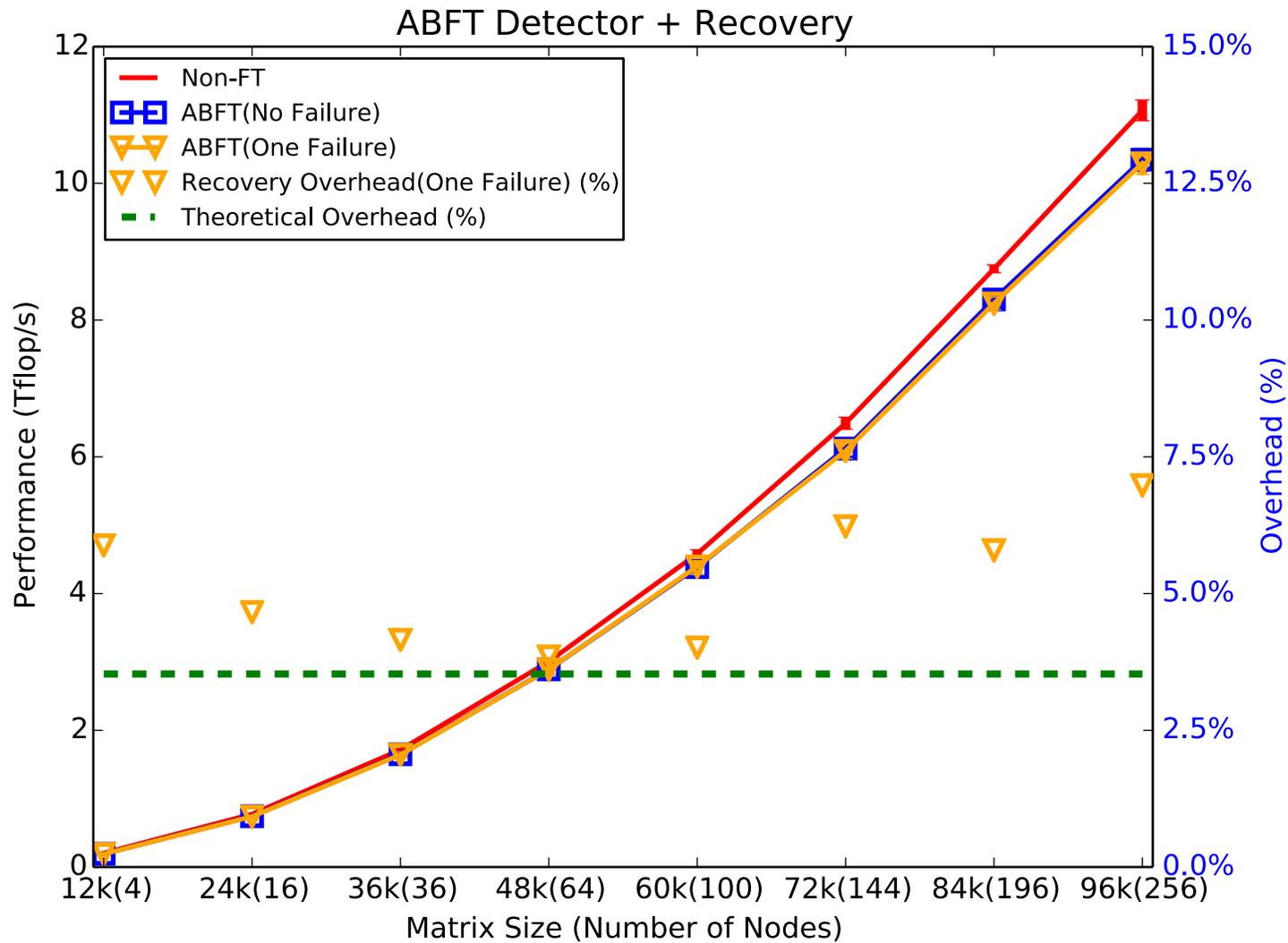
- Experiment 1: Single Bit Flip. ABFT Correction



- Experiment 1: Single Bit Flip. ABFT Correction

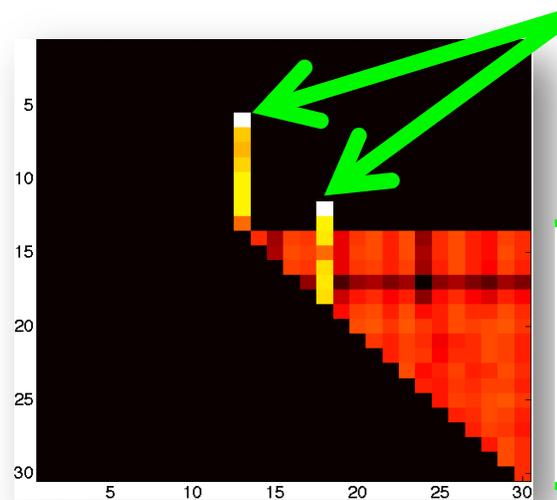


• Experiment 1: Single Bit Flip. ABFT Correction



Experiment Assumed Single Bit Flip, but...

- In a matrix computation an error may propagate.
- In this case need to save inputs and restart the task.

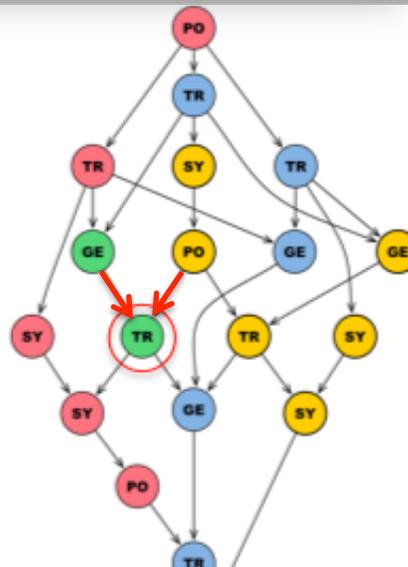


Original errors

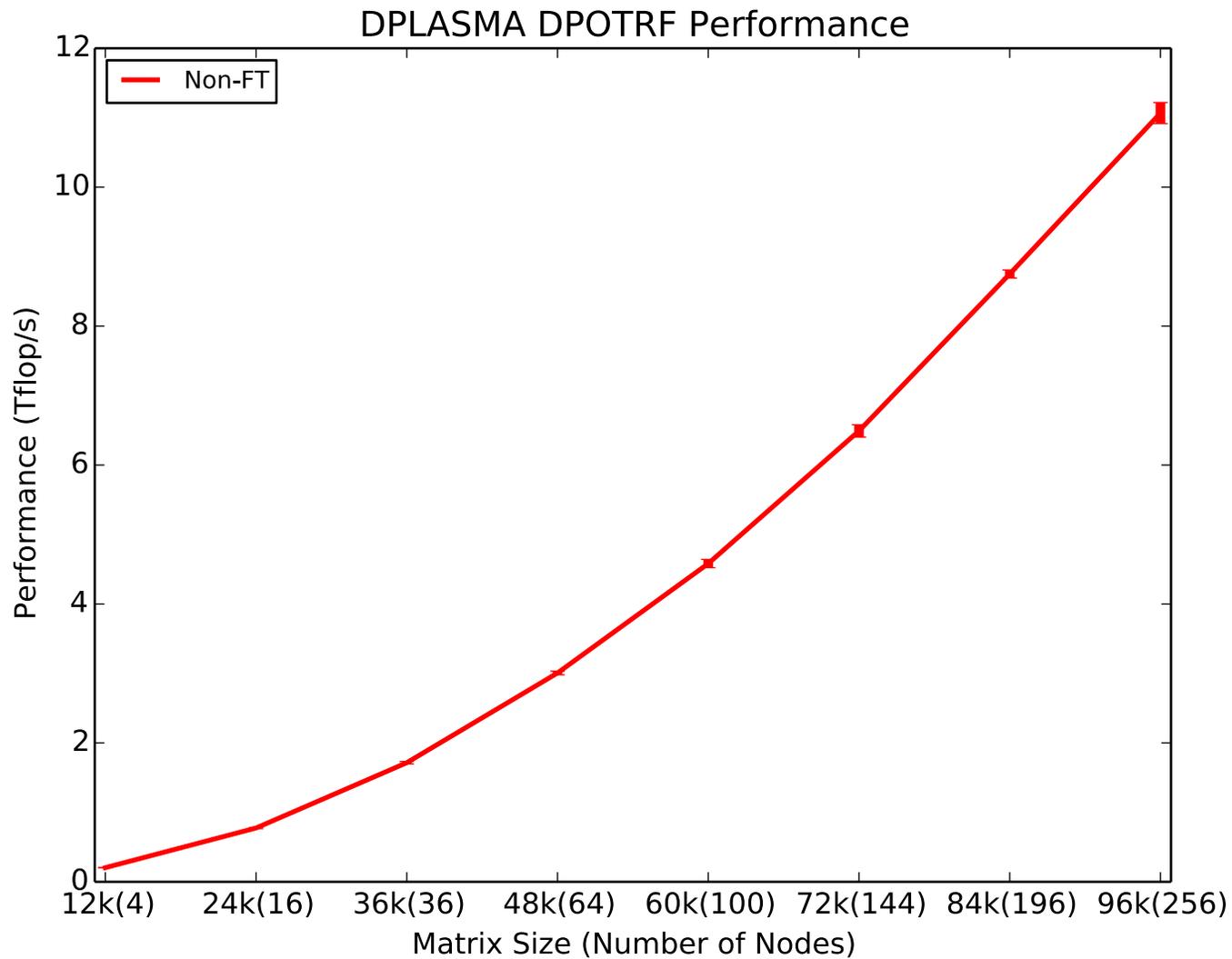
An Analysis of Algorithm-Based Fault Tolerance Techniques, by F. Luk and H. Park, JPDC, Vol 5, No 2, April 1988.

Errors due to propagation

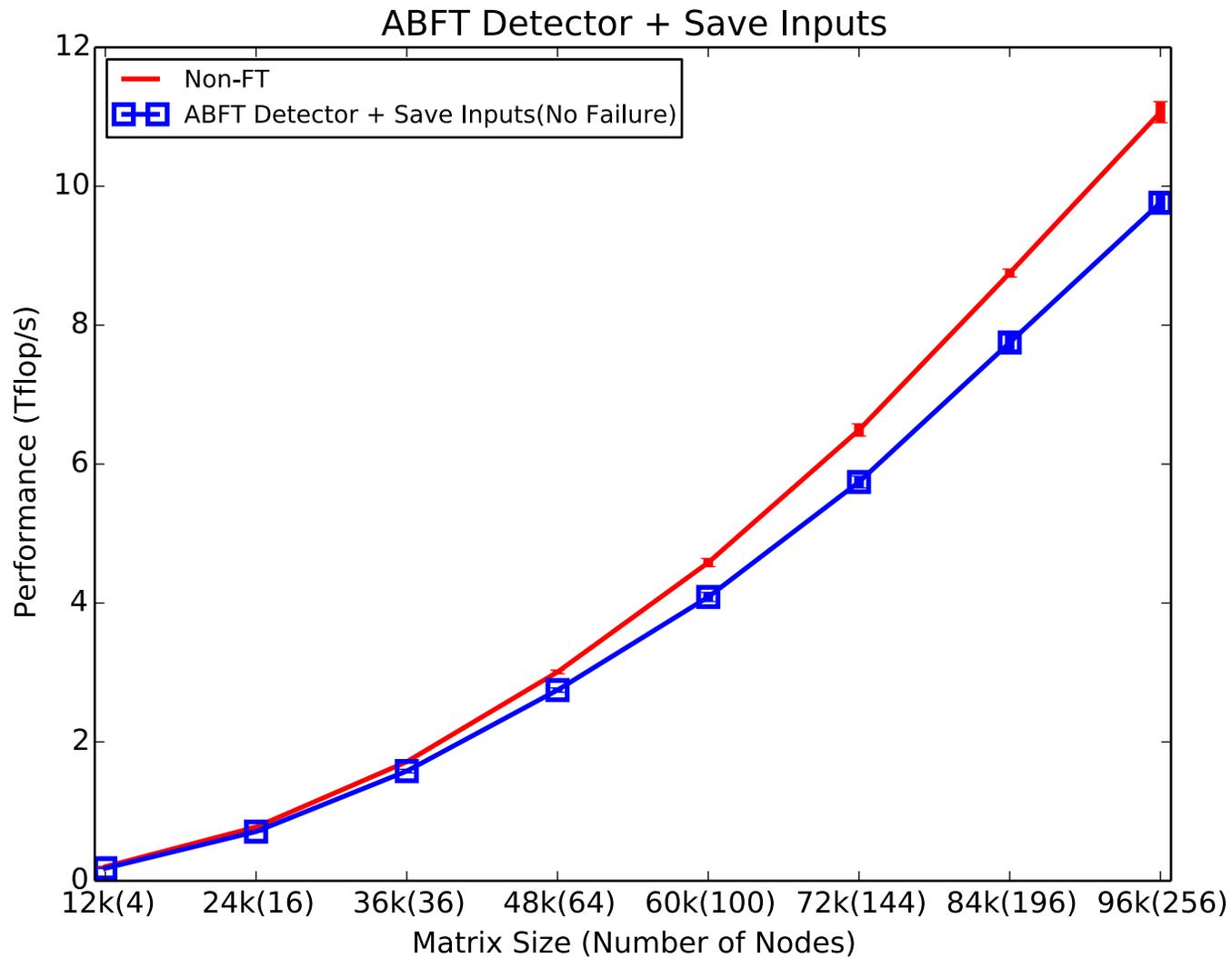
Single bit causes a rank-1 change to the factorization



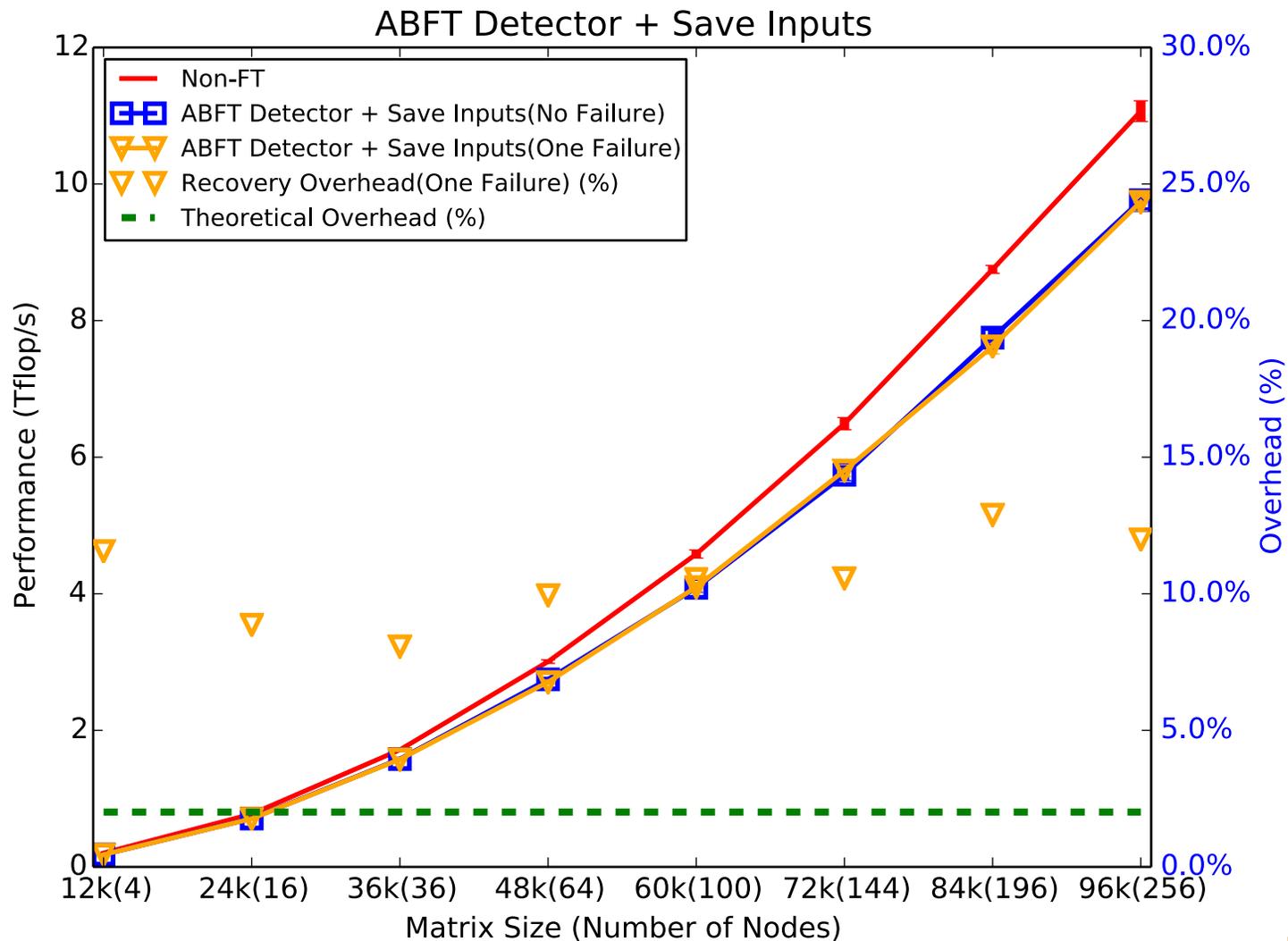
- Experiment 2: Save Task's Inputs Locally and Restart Task



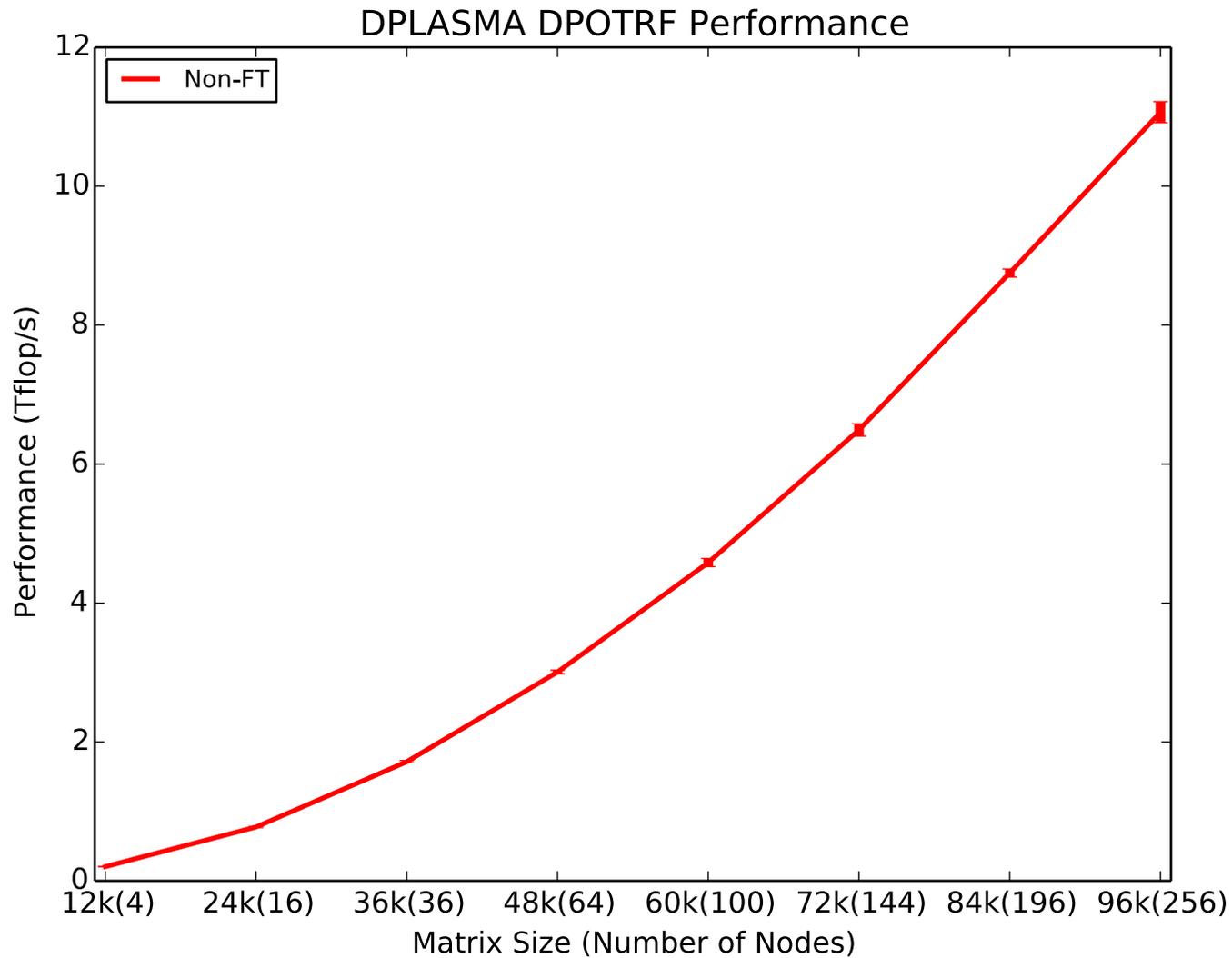
- Experiment 2: Save Task's Inputs Locally and Restart Task



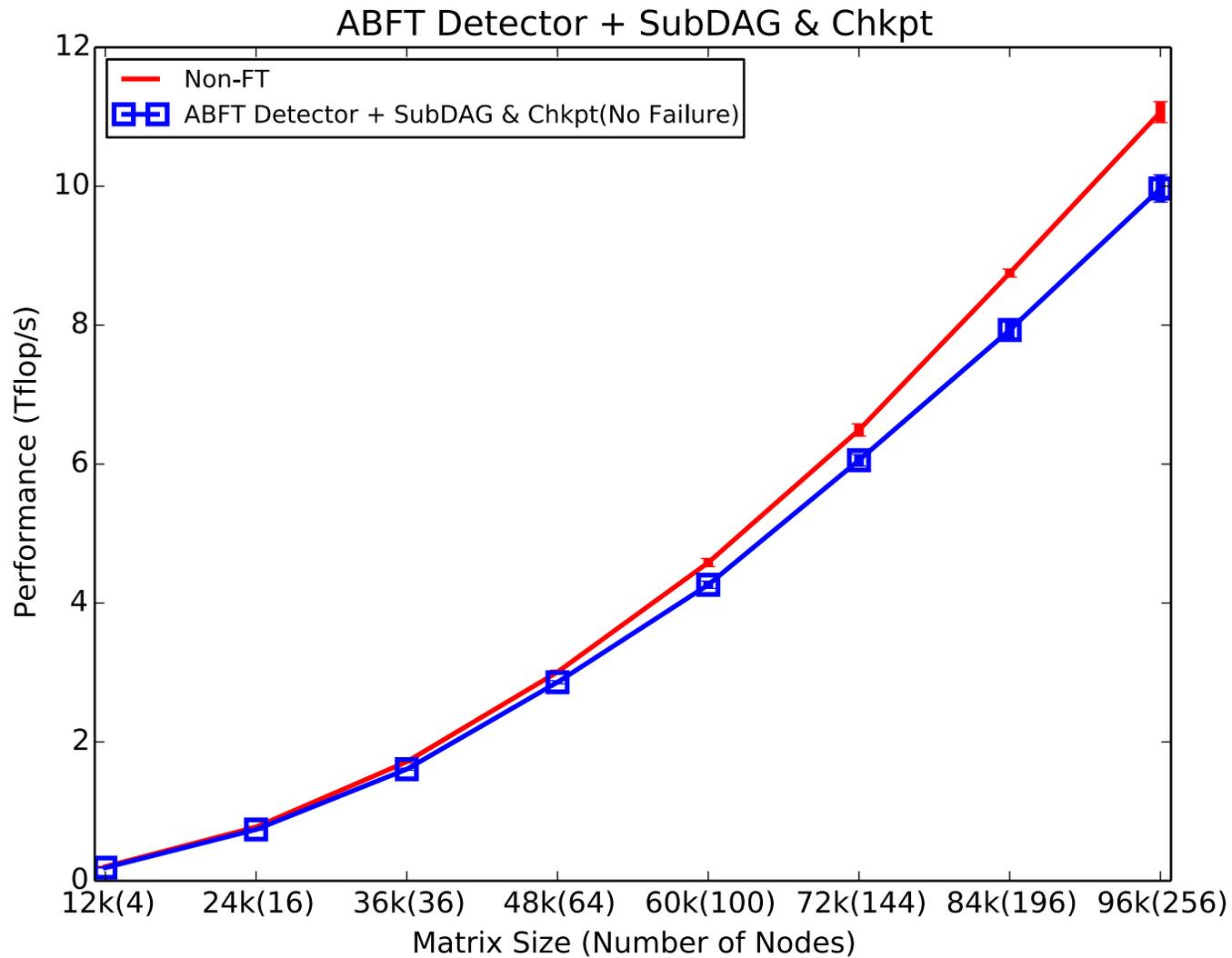
- Experiment 2: Save Task's Inputs Locally and Restart Task



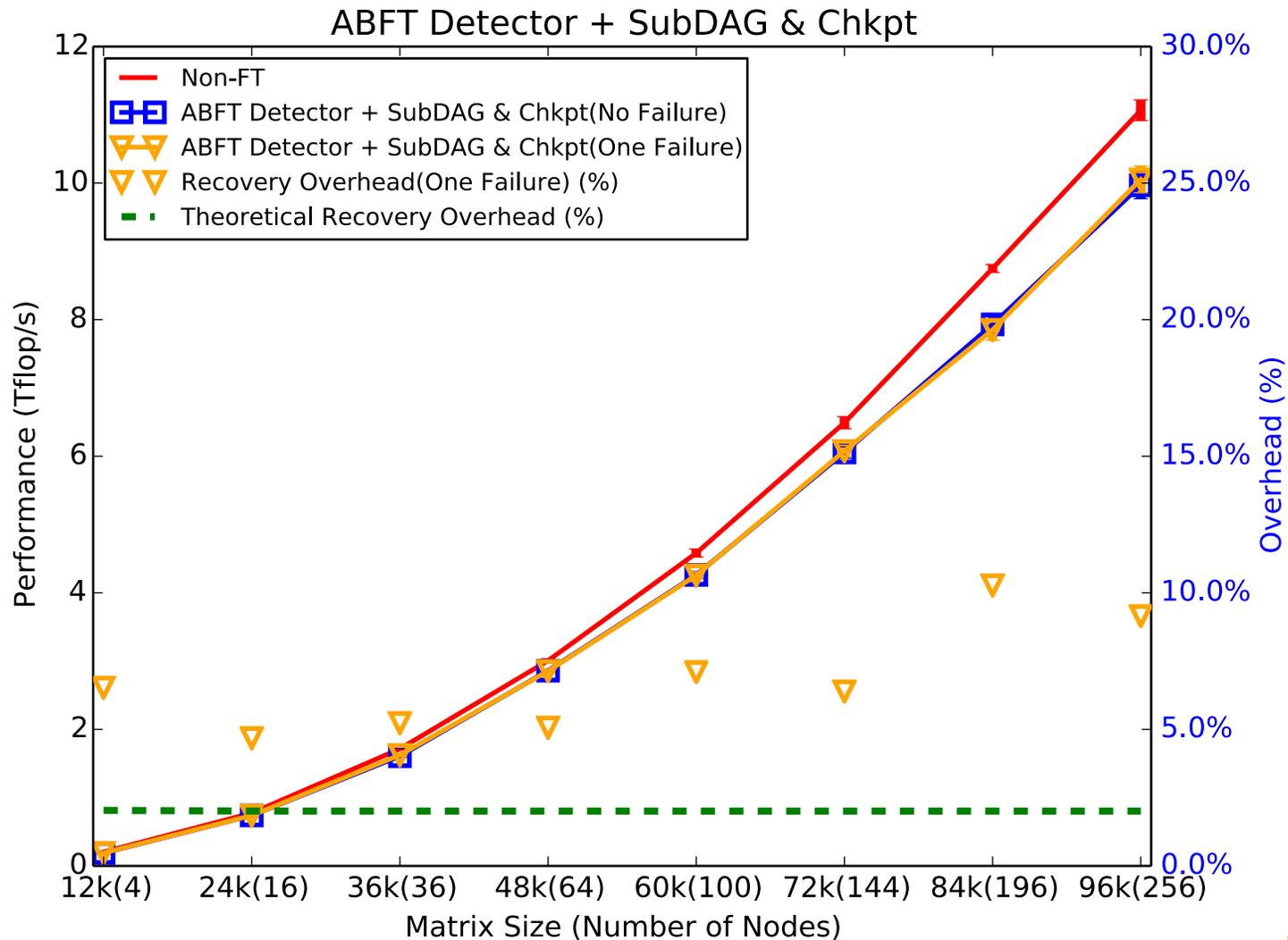
- Experiment 3: Checkpoint every 10 updates



- Experiment 3: Checkpoint every 10 updates



- Experiment 3: Checkpoint every 10 updates



Conclusion & Future work

- *Conclusion*
 - *ABFT enables Software Verification / Validation of computation*
 - *Designing ABFT schemes for specific tasks is easier than designing ABFT schemes for the whole application*
 - *Fairly straightforward to port to other DPLASMA routines*
 - *Two application level mechanisms can be implemented in the runtime, PaRSEC, functions.*
 - *ABFT mechanism has similar idea to extend to other DLA routines.*
- *Future work*
 - Integrate with accelerators
 - Investigate automatic Checkpointing and Rollback-Recovery
 - Extend to support fail stop model (hard error)
- Thanks to George Bosilca, Thomas Herault, Aurelien Bouteiller, Piotr Luszczek, Peng Du, Yulu Jai, and Chongxiao Cao

Dense Factorization

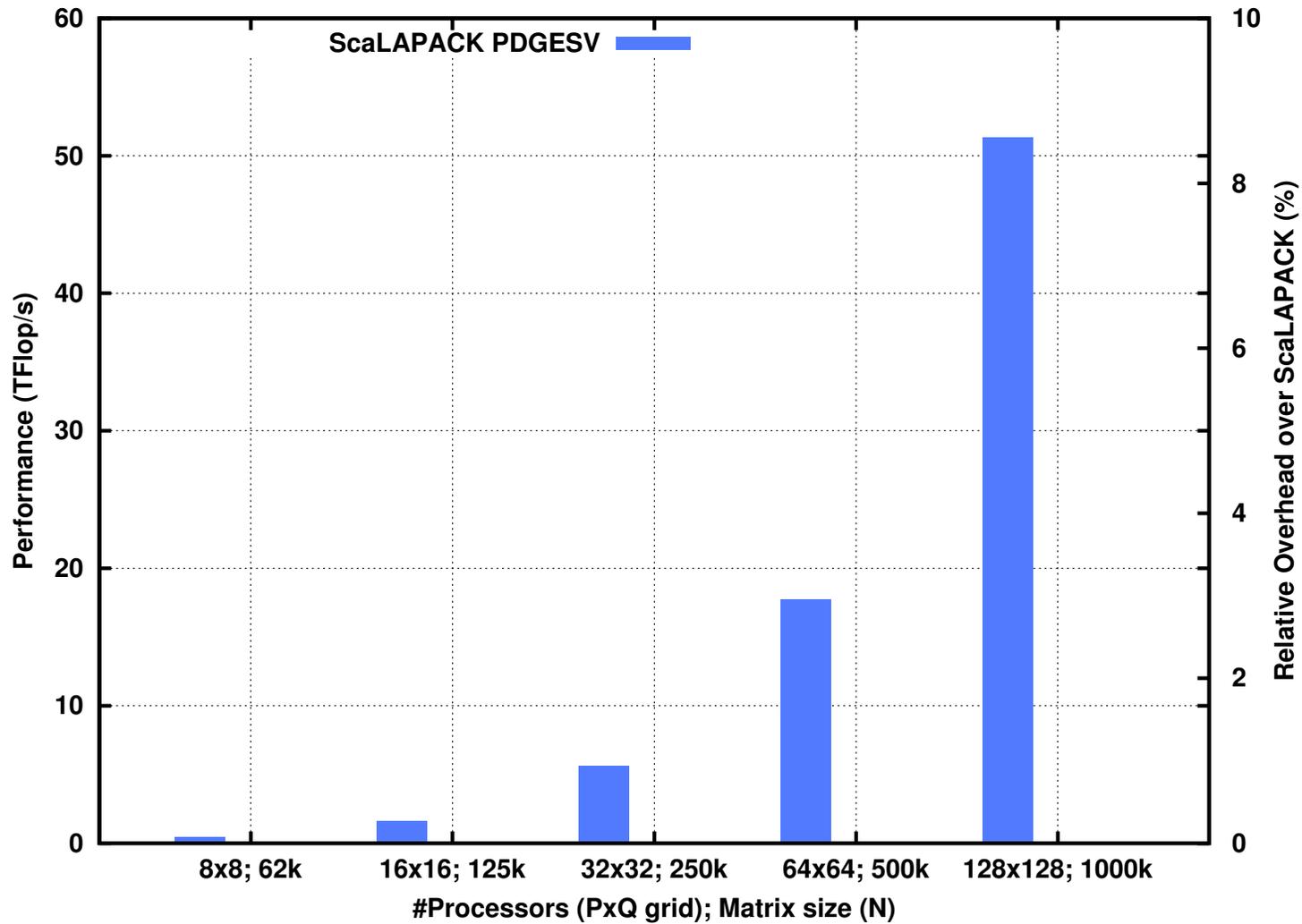
- **Recursive block LU (ScaLapack)**
 - Want to solve $Ax=b$
 - Transform A into LU factorization
 - Solve $Ly=Pb$, then $Ux=y$

$$\begin{bmatrix} L & A & P & A & C & K \\ L & -A & P & -A & C & -K \\ L & A & P & A & -C & -K \\ L & -A & P & -A & -C & K \\ L & A & -P & -A & C & K \\ L & -A & -P & A & C & -K \end{bmatrix}$$

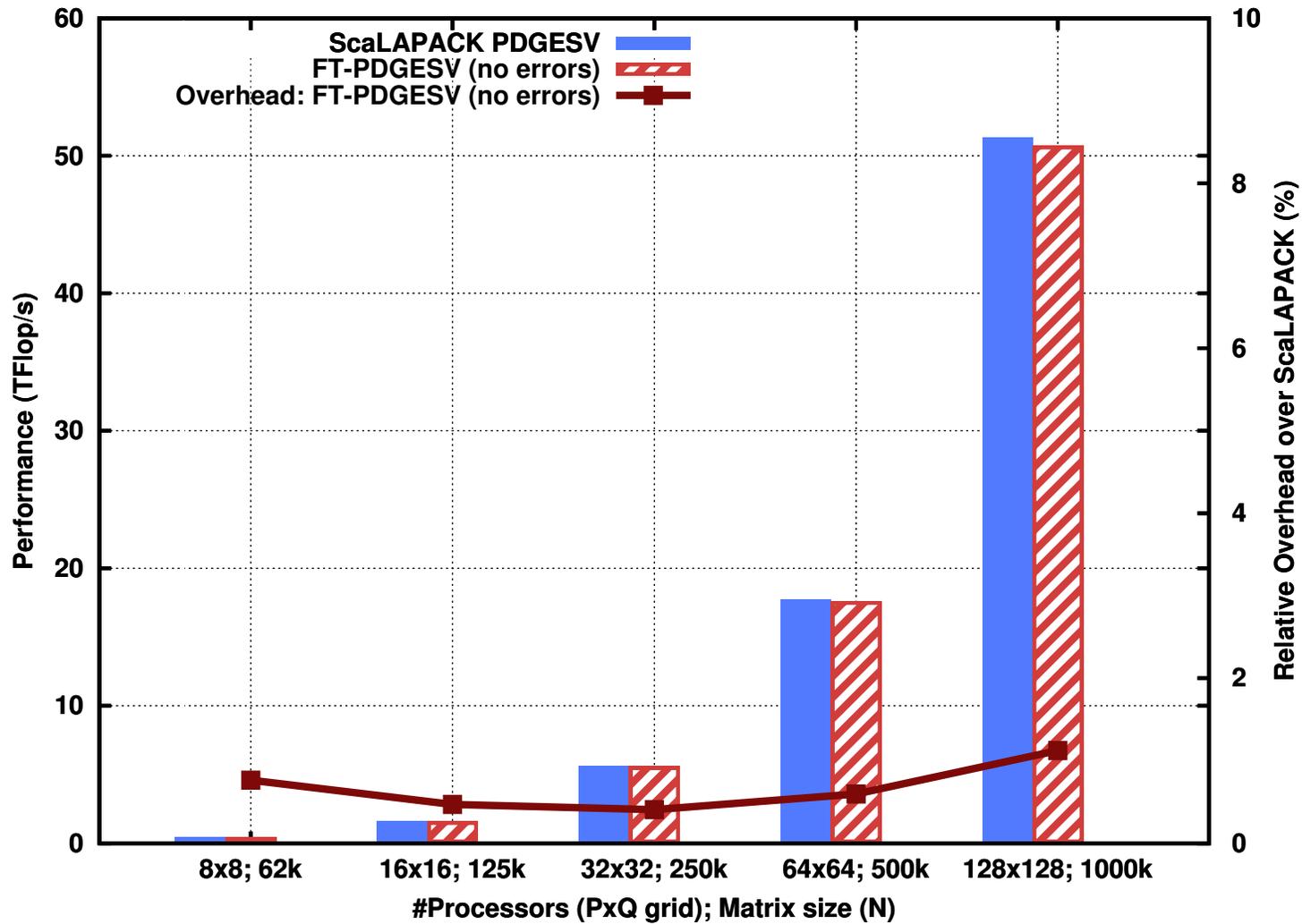


GETF2: factorize a column block

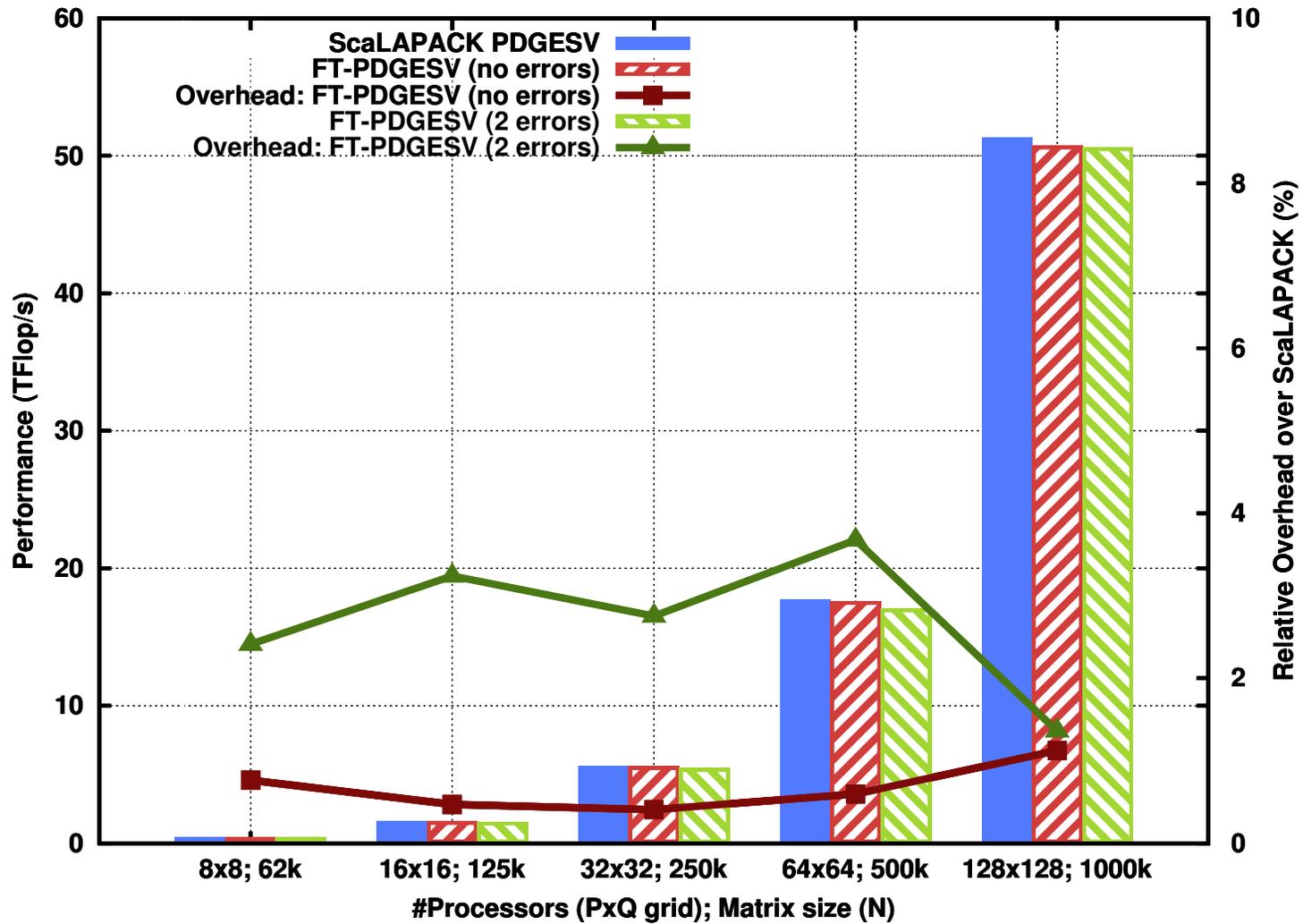
Performance for LU



Performance for LU



Performance for LU

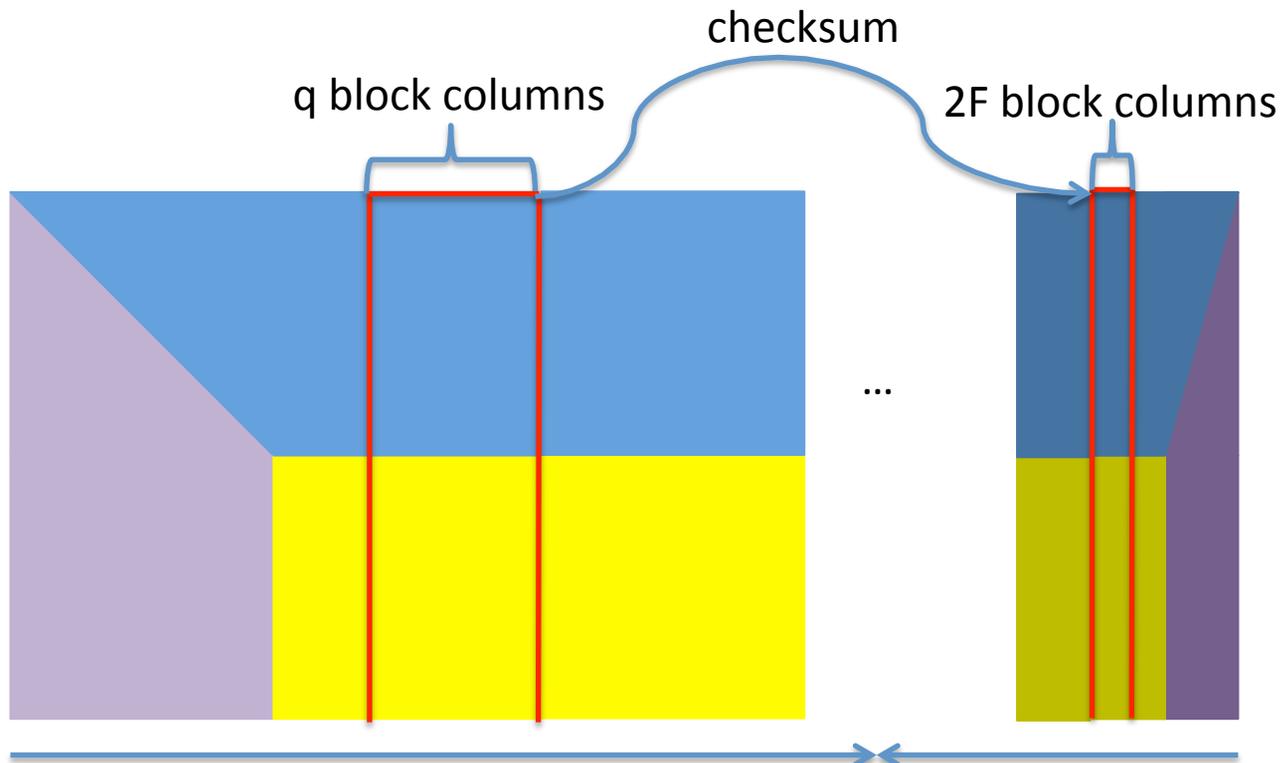


Techniques for Error Protection and Failure Recovery

- **Algorithm-Based Fault Tolerance**
 - **KH Huang & Jacob Abraham, ABFT for Matrix Operations, IEEE Trans. Computers. 01/1984;**
 - Implementation on systolic arrays
 - **Takes advantage of additional mathematical relationship(s)**
 - Already present in algorithm
 - Introduced (cheaply, if possible) by ABFT
 - **Goal of this work is to do an implementation that could be carried out on a complete numerical library.**

Reverse Neighboring Scheme

Elements are really blocks



⋮ The matrix of size $M \times N$,
Blocked in blocks of $mb \times nb$
And distributed over a 2D process grid of $p \times q$
is extended with $\frac{2F \times N}{q \times nb}$ block columns to store a checksum
that will allow to tolerate up to F simultaneous faults on the
same row